UNIVERSIDAD CARLOS III

ESCUELA POLITÉCNICA SUPERIOR

INGENIERÍA TÉCNICA INDUSTRIAL ELECTRÓNICA INDUSTRIAL

PROYECTO FIN DE CARRERA

Implementación Hardware de un test estadístico para aplicaciones criptográficas con un circuito automático de seguridad

AUTOR: Beatriz Guerra Munilla
TUTOR: Anna Vaskova

Curso 2010 – 2011

Agradecimientos

Agradezco a mi familia en especial, por todo el apoyo que me ha dado, a mis amigos por los buenos momentos que me han ayudado a superar todos los obstáculos y a mi tutora, por la infinita paciencia que ha tenido conmigo. Gracias por confiar en mí.

"*Cuando el objetivo te parezca difícil, no cambies de objetivo; busca un nuevo camino para llegar a él.*".

CONFUCIO.

Resumen

Este proyecto propone una opción para un sistema de verificación de algoritmos criptográficos basado en el estudio de la aleatoriedad de una muestra de bits generada. Existen numerosos paquetes de pruebas estadísticas de entre las que se ha elegido el grupo desarrollado por el NIST ya que contiene un software libre del conjunto de test desarrollado en C.

El estudio de esta memoria abarca la implementación hardware de una de estas pruebas con el fin de utilizar las prestaciones de alta velocidad computacional y versatilidad que ofrecen las FPGAs.

Además se desarrolla un sistema de autentificación anti copia para este tipo de tecnología basado en el sistema de identificación DNA y la reconfiguración parcial del diseño.

Palabras clave: FPGA, DNA, reconfiguración, aleatoriedad, P-value, NIST.

Índice de contenidos

Índice de figuras

Índice de tablas

Glosario de términos

Batch Mode: modo consola, es la interfaz de comandos que ofrecen las herramientas del paquete software ISE.
Bit: unidad mínima de información digital (Binary Digit).
Bitstream: es un fichero binario que se utiliza para describir los datos de configuración que se cargaran en una FPGA.
Chip Enable: señal de habilitación de un dispositivo electrónico.
Fabless: tipo de compañía fabricante de semiconductores que carece de una planta de fabricación propia para las obleas de silicio.
Frame: cuadro o parte física de una FPGA.
Generic Map: son los campos genéricos que se pueden definir en distintas arquitecturas mediante VHDL.
Hardware: corresponde a todas las partes tangibles de un sistema informático: sus componentes eléctricos, electrónicos, electromecánicos y mecánicos; sus cables, gabinetes o cajas, periféricos de todo tipo y cualquier otro elemento físico involucrado.
NETs: es el modo de identificación de puertos y señales.
P-value: es un contraste de hipótesis en estadística, está definido como la probabilidad de obtener un resultado al menos tan extremo como el que realmente se ha obtenido (valor del estadístico calculado), suponiendo que la hipótesis nula es cierta. Es fundamental tener en cuenta que el p-valor está basado en la asunción de la hipótesis de partida (o hipótesis nula).
Ratio: tanto por ciento de área física ocupada en la FPGA.
Reset: señal que inicializa un sistema electrónico.
SLICE/M: es un tipo de celda lógica, en este caso tipo M.
Software: equipamiento lógico o soporte lógico de un sistema informático; comprende el conjunto de los componentes lógicos necesarios que hacen posible la realización de tareas específicas.
Test-bench: término usado comúnmente para los bancos de prueba.
Variance: En teoría de probabilidad, la varianza (que suele representarse como σ^2) de una variable aleatoria es una medida de su dispersión definida como la esperanza del cuadrado de la desviación de dicha variable respecto a su media.

Lista de acrónimos

ANSI: American National Standards Institute (Instituto Nacional de Estándares Americano).
ASCII: American Standard Code for Information Interchange (Código Americano Estándar para el Intercambio de Información).
ASIC: Application Specific Integrated Circuit (Circuito Integrado para Aplicaciones Específicas).
CRC: Cyclic Redundancy Check (Comprobación de Redundancia Cíclica).
DNA: Device Number Authentication (Número de Autentificación del Dispositivo).
EDA: Electronic Design Automation (Diseño electrónico Automatizado).
FPGA: Field Programmable Gate Array (Vector de Compuertas de Campo Programables).
FSM: Finite State Machine (Maquina de Estados Finita).
GUI: Graphic User Interface (Interfaz Gráfica de Usuario).
ISE: Integrated Software Environment (Entorno de Software Integrado).
JTAG: Joint Test Action Group (Acción de Grupo de Pruebas Conjuntas).
LED: Light Emitting Diode (Diodo Emisor de Luz).
NIST: National Institute of Standards and Technology (Instituto Nacional de Estándares y Tecnología).
PC: Personal Computer (Computador Personal).
PRNG: Pseudo-Random Number Generator (Generador de Números Pseudo Aleatorios).
RAM: Random-Access Memory (Memoria de Acceso Aleatorio).
RNG: Random Number Generator (Generador de Números Aleatorios).
ROM: Read-Only Memory (Memoria de Solo Lectura).
SRL: Shift Register Left (Registro de Desplazamiento a la Izquierda).
UCF: User Constraints File (Fichero de Restricciones del Usuario).
VHDL: VHSIC Hardware Description Language (Lenguaje de Descripción de Hardware de VHSIC).
VHSIC: Very High Speed Integrated Circuit (Circuito Integrado de Muy Alta Velocidad).

1. Introducción.

1.1. Planteamiento inicial.

Hoy en día la seguridad es un tema de gran preocupación para nuestra sociedad. La tecnología avanza rápidamente y los diseños electrónicos deben ser robustos y resistentes ante los distintos tipos de falsificación como: la ingeniería inversa, la clonación, la reconstrucción o la manipulación de los productos. Estos productos falsificados dañan y perjudican seriamente los beneficios de la Industria y la confianza del cliente en los servicios ofrecidos por ésta.

Ante tales amenazas, una solución es proteger los datos mediante el uso de técnicas criptográficas para lo que es necesario verificar la calidad de los generadores de números aleatorios y pseudo-aleatorios en los cuales se basan en este tipo de técnicas.

Existen varios métodos para probar la aleatoriedad de estos generadores. El NIST (National Institute of Standards and Technology) unificó en un documento una serie de tests estadísticos para dicho fin, atendiendo a distintos tipos de no-aleatoriedad que pueden existir en una secuencia de bits. Este conjunto de pruebas no es excluyente de la necesidad de un criptoanálisis pero sí se trata de un sistema de descarte inicial.

En este proyecto se ha realizado la implementación hardware de un test estadístico como parte del trabajo completo de implementación de todos los tests estadisticos del NIST. El trabajo que ha sido el vínculo para el análisis de las pruebas estadísticas es "Aceleración Hardware De Tests Estadísticos para evaluación de Algoritmos Criptográficos" que ha sido presentado como Tesis de Máster este año y publicado en diferentes conferencias. Además se incluye un eficaz sistema de seguridad anti copia. Este sistema de seguridad automático es una aplicación independiente, que se puede añadir para proteger cualquier otro diseño y aporta una solución rápida para evitar la clonación directa.

Según el Artículo I de Ley de propiedad Intelectual: "*La propiedad intelectual de una obra literaria, artística o científica corresponde al autor por el solo hecho de la creación* ", por lo que resulta de vital importancia el estudio sobre algoritmos útiles en la protección de este derecho (test para generadores de números aleatorios), y a su vez la protección de dichos avances y estudios (sistema de seguridad anti copia).

1.2. Objetivos del Proyecto.

El objetivo fundamental del proyecto es diseñar una herramienta segura, rápida, sencilla y de bajo coste capaz de verificar la aleatoriedad de los distintos tipos de algoritmos criptográficos, generadores de números aleatorios, etc.

- Segura por la incorporación de una aplicación de seguridad automática en el arranque que impide la copia fácil del diseño, y garantiza la fiabilidad original.
- Rápida, ya que al tratarse de una implementación a un nivel más físico (Hardware) se evita la carga de sistemas más complejos (Software), obteniendo una gran ventaja en la velocidad de procesamiento.
- Sencilla, gracias a la versatilidad del medio en que se implementa, las FPGAs.
- De bajo coste, ya que se ha implementado sobre uno de los FPGA de nivel medio, con el fin de optimizar el diseño y reducir gastos.

En base a este objetivo principal surge un segundo objetivo respecto al circuito de seguridad. Se pretende conseguir que el diseño de la aplicación añadida de anti clonación sea independiente y capaz de acoplarse a cualquier otro diseño realizado en FPGAs para utilizarlo en futuras implementaciones.

2. Estado del Arte.

Tecnología de aplicación y configuración.

La tecnología en la que se basa el diseño son las FPGAs dada la facilidad para implementar circuitos mediante un lenguaje de descripción hardware (VHDL), que permite crear aplicaciones del diseño electrónico digital. Este tipo de dispositivos, de uso cada vez más extendido en el mercado actual, resulta una solución de bajo coste y gran flexibilidad en el desarrollo de nuevos productos electrónicos.

Los circuitos integrados están presentes en gran cantidad de productos industriales, la alternativa que ofrece la FPGA permite realizar un circuito integrado a medida sin los riesgos económicos asociados a otras tecnologías menos flexibles. Hoy en día forman parte de sistemas de automoción, electrónica de consumo, etc. Esta tecnología tiene aplicación en todas las industrias que requieren computación a alta velocidad.

Las FPGAs tienen la ventaja de ser reprogramables (lo que añade una enorme flexibilidad al flujo de diseño), sus costes de desarrollo y adquisición son mucho menores para pequeñas cantidades de dispositivos y el tiempo de desarrollo es también menor. La lógica programable puede reproducir desde funciones tan sencillas como las llevadas a cabo por una puerta lógica o un sistema combinacional hasta complejos sistemas en un chip.

Los FPGAs se utilizan en aplicaciones similares a los ASIC, aunque son más lentos y tienen un mayor consumo de potencia, hay que tener en cuenta su alta inmunidad a interferencias electromagnéticas.

Cualquier circuito de aplicación específica puede ser implementado en un FPGA, siempre y cuando esta disponga de los recursos necesarios. Las aplicaciones donde más comúnmente se utilizan los FPGA incluyen a los DSP (procesamiento digital de señales), radio definido por software, sistemas aeroespaciales y de defensa, prototipos de ASICs, sistemas de imágenes para medicina, sistemas de visión para computadoras, reconocimiento de voz, emulación de hardware de computadora, entre otras. Cabe notar que su uso en otras áreas es cada vez mayor, sobre todo en aquellas aplicaciones que requieren un alto grado de paralelismo, donde se requieren múltiples interfaces digitales o con muchas entradas y salidas, como las comunicaciones de alta velocidad de transmisión de datos.

Xilinx es la mayor empresa en investigación y desarrollo de chips FPGAs. Fue fundada por Ross Freeman (el inventor de las FPGA), Bernie Vonderscmitt (pionero del

concepto de compañía *fabless*) y Jim Barnett en 1984 y con base en Silicon Valley. Al año siguiente desarrollaron su primera FPGA, el modelo XC2064. Las familias de dispositivos de Xilinx son: glue logic (CoolRunner y CoolRunner II), bajo coste (Spartan) y alto rendimiento (Virtex).

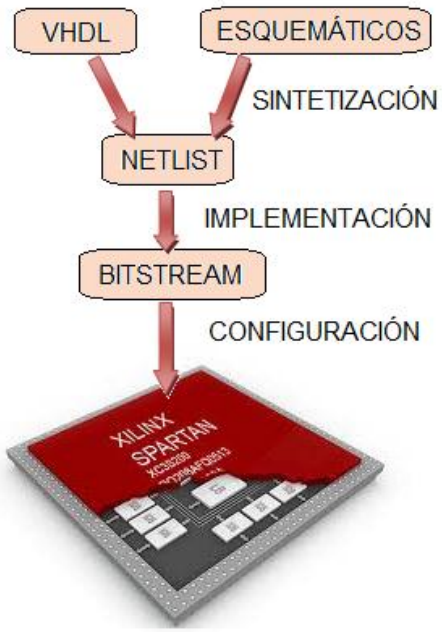
Otro de los fabricantes mayoritarios de FPGAs es Altera que ofrece otros modelos referentes en alto rendimiento (Stratix) y bajo coste (Cyclone).

Los sistemas electrónicos reconfigurables del tipo FPGA son dispositivos de alta complejidad que no sería abarcable sin la ayuda de un entorno con herramientas que asistan en el proceso de diseño, simulación, síntesis del resultado y configuración del hardware. Un ejemplo de un entorno de este tipo es el software de la empresa Xilinx denominado ISE.

Este software constituye un verdadero entorno de diseño automático (EDA). La interfaz gráfica de usuario GUI se denomina Proyect Navigator y facilita el acceso a todos los componentes del proyecto.

Los diseños de usuario se pueden introducir mediante diferentes formatos. Los más utilizados son: los esquemáticos, los grafos de estados y las descripciones hardware en VHDL. Una vez compilados los diseños se puede simular su comportamiento a nivel funcional o a nivel temporal. A nivel funcional no tiene en cuenta los retardos provocados por el hardware y a nivel temporal simula el diseño teniendo en cuenta cómo se va a configurar el hardware. El proceso de diseño e implementación de la FPGA se estructura en una serie de pasos:

1. Diseño.
   - Lenguaje HDL.
   - Esquemáticos del circuito.
2. Síntesis del diseño.
   - Generación del Netlist.
3. Implementación del diseño.
   - Translate, Map, Place&Route.
4. Configuración de la FPGA.
   - Generación del bitstream.



VHDL representa la combinación de VHSIC y HDL, donde VHSIC es el acrónimo de "Very High Speed Integrated Circuit" (circuitos integrados de muy alta velocidad) y HDL es a su vez el acrónimo de "Hardware Description Language" (Lenguaje de descripción hardware).

EL lenguaje VHDL es un lenguaje que fue patrocinado por el departamento de defensa de los Estados Unidos dentro del programa VHSIC. En 1987 fue aprobado como estándar IEEE (bajo la norma IEEE 1076) y a partir de aquí comienza su consolidación como lenguaje estándar internacional y su influencia en los métodos de diseño de sistemas digitales. En VHDL hay varias formas con las que podemos diseñar el mismo circuito y es tarea del diseñador elegir la más apropiada.

- Funcional: Describimos la forma en que se comporta el circuito. Esta es la forma que más se parece a los *lenguajes de software* ya que la descripción es secuencial. Estas sentencias secuenciales se encuentran dentro de los llamados procesos en VHDL. Los procesos son ejecutados en paralelo entre sí, y en paralelo con asignaciones concurrentes de señales y con las instancias a otros componentes.
- Flujo de datos: describe asignaciones concurrentes (en paralelo) de señales.
- Estructural: se describe el circuito con instancias de componentes. Estas instancias forman un diseño de jerarquía superior, al conectar los puertos de estas instancias con las señales internas del circuito, o con puertos del circuito de jerarquía superior.
- Mixta: combinación de todas o algunas de las anteriores.

En VHDL también existen formas metódicas para el diseño de máquinas de estados, filtros digitales, bancos de pruebas etc.

Aunque ambas empresas ofrecen prestaciones similares, se ha decidido trabajar con la FPGA Spartan 6 de Xilinx ya que posee varias de las características necesarias para el diseño, como por ejemplo sistemas de protección contra la duplicación sin autorización.

El bitstream que define la funcionalidad de la FPGA se carga en el dispositivo durante la configuración. Consecuentemente, esto implica que una compañía sin escrúpulos puede capturar el fichero binario y crear una copia desautorizada del diseño.

Existen múltiples técnicas para proteger este fichero binario de configuración de la FPGA y los núcleos de propiedad intelectual integrados en la FPGA. Una de las técnicas es llamada autentificación, la cual usa el valor único DNA del dispositivo. En este proyecto se ha elegido usar este tipo de técnico debido a que ofrece una posibilidad de acceso libre con el paquete básico y gratuito que ofrece Xilinx, y cuya aplicación se puede desarrollar en otros dispositivos FPGA distintos[1].

Además existen otro tipo de protecciones para varios modelos de la FPGA Spartan 6 como los modelos XC6SLX75/T, XC6SLX100/T, and XC6SLX150/T que tienen también una lógica de encriptación AES para el chip ofreciendo un alto nivel de seguridad en el diseño. Sin el conocimiento de la clave de encriptación no se puede analizar externamente el bitstream para entender o clonar el diseño.

Esta lógica de encriptación AES del dispositivo no puede ser usada para cualquier otra aplicación excepto la descripción del bitstream. El algoritmo de encriptación AES es una norma oficial del NIST and del Departamento de Comercio de Estados Unidos [2].

Otro de los aspectos destacables de las FPGAs aparte de que es reprogramable, es su capacidad de reconfiguración. Como el nombre propio indica, este aspecto interviene críticamente a la fase de configuración de la FPGA. Existen por tanto diversos métodos para la reconfiguración parcial de estos dispositivos.

La Reconfiguración parcial consiste en cambiar la configuración de una parte de la FPGA sin que deje de funcionar el resto de ella. Otra posible aplicación podría ser la utilización de comunicaciones reconfigurables o sistemas criptográficos, que permitiría cambios de cifrado hardware o incluso de forma de comunicación manteniendo el mismo circuito y a distancia. Es además una ventaja para el trabajo con las propias *FPGAs*, puesto que reconfigurar parcialmente tarda menos tiempo que configurar la *FPGA* con un *bitstream* completo, y el *bitstream* parcial ocupa menos [3].

Además, esta reconfiguración parcial puede realizarse desde un sistema instalado dentro de la FPGA (microprocesador), en lo que se conoce como auto-reconfiguración parcial, de tal manera que la FPGA se reconfigura a sí misma.

Para reconfigurar la FPGA con este método se necesita un circuito “inteligente”, que sea capaz de decidir el momento preciso y cargar la configuración adecuada en la memoria de la FPGA, por lo que normalmente se utiliza un ordenador. Sin embargo, la reconfiguración puede ser realizada por un dispositivo implementado en la propia FPGA, sin que sea necesario ningún microprocesador externo adicional ni ordenador, esta técnica es llamada auto reconfiguración parcial. La utilización de Soft-Core Processors (SCP) como dispositivos encargados de realizar la reconfiguración de una parte de la FPGA, añade grandes posibilidades a esta técnica. Gracias a la auto-reconfiguración estos SCPs pueden reprogramar una parte de la FPGA con el coprocesador o hardware elegido, obteniendo una mejora de rendimiento apreciable y circuitos hardware capaces de re-adaptarse, evolucionar o auto-repararse.

Se definen distintos tipos de reconfiguraciones [4]:

1. Reconfiguración parcial: configuración de un cierto número de celdas de la FPGA sin borrar la memoria de configuración, ni apagar y encender la FPGA. Se distinguen dos zonas de la FPGA:

   - Área Reconfigurable: zona de la FPGA que va a ser reconfigurada.
   - Área Estática o No Reconfigurable: zona de la FPGA que no se va a modificar.

2. Reconfiguración parcial estática: reconfiguración parcial de la FPGA parando el funcionamiento de la parte estática o no reconfigurable. Esto impide obviamente la auto-reconfiguración. Existen dos tipos:

   - Reconfiguración parcial estática sin conservar el estado: se reconfigura sin mantener el estado de la FPGA (luts, flip-flops, máquinas de estado, etc. se resetean)
   - Reconfiguración parcial estática conservando el estado: se mantiene el estado del resto de la FPGA al realizar la reconfiguración.

3. Reconfiguración parcial dinámica (o activa): reconfiguración parcial de una parte de la FPGA sin parar el funcionamiento del resto de ella, con el consiguiente peligro de cortocircuito si no se toman las medidas necesarias.

4. Auto-Reconfiguración parcial: un microprocesador embebido se comunica con la memoria de programación de la FPGA y la reconfigura. El microprocesador debe formar parte del área estática (aunque sus periféricos no tienen por qué).

Los métodos anteriores de reconfiguración basan su funcionamiento en cambios realizados en el bitstream mediante la implementación de una parte de menor tamaño del circuito. Este proceso requiere la utilización o comunicación con un PC que se encargue de re sintetizar e implementar el bitstream en diferencias, o una memoria que guarde distintos ficheros binarios para diferentes tipos de configuración. Una de las opciones que existen es la modificación del bitstream sin la necesidad de una re-implementación, trabajando directamente sobre el fichero binario. Este tipo de método es el que se usa en la ingeniería inversa cuando se trata de extraer un diseño lógico a partir de la cadena de unos y ceros del fichero de configuración de la FPGA. En la mayoría de los diseños en FPGAs se genera un bitstream que no se ha protegido con un método de encriptado con algoritmos como el AES. Este fichero posee una relación plana entre los datos binarios y el circuito diseñado. Por ende, uno de los métodos de reconfiguración es modificar un bitstream plantilla o base y cambiar las partes del diseño que se necesitan variar.

Métodos estadísticos para el estudio de la aleatoriedad.

La necesidad de obtener números aleatorios y pseudo-aleatorios se plantea en muchas aplicaciones criptográficas, pues se emplean llaves que deben ser generadas con dichas características. Por ejemplo, para cantidades auxiliares usadas en generación de firmas digitales, o para generar desafíos en autentificación de protocolos. El Instituto Nacional de Estándares y Tecnología (NIST) proporciona un conjunto de pruebas estadísticas de aleatoriedad y considera que estos procedimientos son útiles en la detección de desviaciones de una secuencia binaria en la aleatoriedad. Existen dos tipos básicos de generadores usados para producir secuencias aleatorias: Generadores de Números Aleatorios (RNGs) y Generadores de Numeras Pseudo-Aleatorios (PRNGs).

Para aplicaciones criptográficas, ambos tipos de generadores producen un flujo de ceros y unos que pueden ser divididos en sub-flujos o bloques de números aleatorios. . Los primeros usan como entrada una fuente no determinista (p.e. ruido de un circuito eléctrico) mientras que los segundos generan una secuencia de bits bastante aleatoria en base a una semilla del generador. El interés del proyecto está en la revisión de un generador tipo PRNGs, en este caso, si la semilla es desconocida, en el paso siguiente el número producido en la secuencia debe ser impredecible a pesar de todo conocimiento de números aleatorios anteriores en la secuencia.

Las aplicaciones criptográficas exigen que las secuencias pseudo-aleatorias sean indistinguibles de las verdaderamente aleatorias. La aleatoriedad es una propiedad que puede ser descrita en términos de probabilidad ya que no se dispone de ninguna prueba matemática que determine de forma categórica la propiedad de aleatoriedad de una

secuencia de bits dada. Las siguientes suposiciones son hechas con respecto a las secuencias aleatorias binarias a comprobar:

- Uniformidad: en cualquier punto en la generación de una secuencia de bits aleatorios o pseudo-aleatorios, la ocurrencia de un '0' o '1' es igualmente probable; por ejemplo, la probabilidad de cada uno es exactamente ½=0,5. El número de ceros o unos esperados es n/2, donde n=es la longitud de la secuencia.
- Escalabilidad: cualquier test aplicable a una secuencia, también puede ser aplicado a subsecuencias extraídas al azar. Si una secuencia es aleatoria, entonces cualquier subsecuencia extraída semejante debe ser también aleatoria. Por lo tanto, cualquier subsecuencia extraída debe pasar cualquier test de aleatoriedad.
- Consistencia: el funcionamiento de un generador debe ser consistente a través valores iniciales (semilla). Es insuficiente probar un PRNG basado en la salida de un simple valor inicial (semilla), o una RNG en base de una única salida producida por una salida física.

Un test estadístico es desarrollado para probar una Hipótesis nula específica (H0). Para esta serie de tests, la hipótesis nula es que la secuencia es aleatoria, mientras que la Hipótesis alternativa (Ha) es que no lo es. La conclusión de la prueba consiste en aceptar o denegar la H0.

A partir de una distribución de referencia se determina un valor crítico. Durante una prueba, un valor de la prueba estadística se calcula sobre los datos (la secuencia se está probando). Este valor estadístico de la prueba se compara con el valor crítico. Si el valor estadístico de la prueba excede el valor crítico, la hipótesis nula de aleatoriedad es rechazada. De lo contrario, la hipótesis nula (la hipótesis de aleatoriedad) no se rechaza (es decir, la hipótesis es aceptada).

Si en la conclusión se deniega la H0, es decir, se demuestra que la secuencia no es aleatoria, se denomina que existe un error de Tipo I. La probabilidad de este error se denomina nivel *de significancia* y se denota con α. Si por el contrario, se parte de una hipótesis nula de que la muestra es no aleatoria y se demuestra que lo es, se produce un error de Tipo II y la probabilidad de que ocurra se denota por β.

	Conclusión	
Situación real	**Se acepta H0**	**Se acepta Ha**
Dato aleatorio (H0 es verdadera)	No hay error	Error Tipo I
Dato no aleatorio (Ha es verdadera)	Error Tipo II	No hay error

Tabla de errores Tipo I y II.

El test estadístico es usado para calcular un *P-value* que muestra la fuerza de la evidencia en contra de la H0. La probabilidad de que el teste estadístico elegido tome valores iguales o peores que el valor observado cuando se considera la hipótesis nula se denomina *P-value.* Si este valor es igual a 1, la secuencia de estudio tendría una perfecta aleatoriedad. El *P-value* se puede calcular a partir de diferentes funciones matemáticas.

Para $\alpha = 0.001$ se puede afirmar con un 99,9% de seguridad que si:

- $P - value < \alpha$; H0 es falsa, es decir, la secuencia no es aleatoria.
- $P - value \geq \alpha$; H0 es verdadera y la secuencia es aleatoria.

Para $\alpha = 0.01$ se pueden realizar las mismas afirmaciones con un 99% de fiabilidad.

Existen numerosos test estadísticos que determinan la presencia o ausencia de patrones en una secuencia de bits, capaces de indicar si existe no-aleatoriedad. El conjunto de pruebas de NIST [5] es un paquete estadístico que consiste en 15 pruebas que se desarrollaron para probar la aleatoriedad de (arbitrariamente largas) secuencias binarias producidas por hardware y software basado en generadores criptográficos de números aleatorios o pseudo-aleatorios. Dichas pruebas se enfocan en diversos tipos de no aleatoriedad que pueden existir en una secuencia. Las 15 pruebas son:

1. Prueba de frecuencia (Monobit): esta prueba mide la proporción de ceros y unos de toda una secuencia.
2. Prueba de frecuencia dentro de un bloque: esta prueba mide la proporción de unos dentro de un bloque de M bits.
3. Prueba de corriente: esta prueba mide el total de corrientes en una secuencia, donde una corriente es una secuencia interrumpida de bits idénticos.
4. Prueba de la más larga corriente de unos en un bloque: Esta prueba mide la corrida más larga de unos dentro de un bloque de M bits.
5. Prueba de rango de la matriz binaria: esta prueba mide el rango de sub-matrices disjuntas de toda la secuencia.
6. Prueba de la transformada discreta de Fourier (Espectral): esta prueba mide las alturas de los picos en las transformadas discretas de Fourier de las secuencias.
7. Prueba de la no acumulación de coincidencia de plantilla: esta prueba mide el número de ocurrencias de cadenas de destino especificadas. Una ventana de m bits es usada para buscar un patrón específico de m bits.
8. Prueba de acumulación de coincidencia de plantilla: esta prueba también mide el número de ocurrencias de cadenas destino pre-especificadas. La diferencia con la prueba anterior reside en la acción realizada al encontrar un patrón.
9. Prueba de Estadística Universal de Maurer: esta prueba mide el número de bits entre los patrones de juego (una medida que está relacionada con la longitud de una secuencia comprimida).
10. Prueba de complejidad lineal: esta prueba mide la longitud de un Registro de Desplazamiento con Retroalimentación Lineal (LFSR). Una baja longitud LFSR implica no aleatoriedad.
11. Prueba de serie: esta prueba mide la frecuencia de todos los posibles patrones de m bits acumulados a través de la secuencia completa.
12. Prueba de entropía aproximada: esta prueba tiene el mismo enfoque que la anterior, con el propósito de comparar la frecuencia de bloques acumulados de dos consecutivas/adyacentes longitudes (m y m + 1).

13. Prueba de sumas acumulativas: esta prueba mide la excursión máxima (desde cero) del paseo aleatorio definido por la suma acumulada de ajustados (-1, +1) dígitos en la secuencia.
14. Prueba de excursiones aleatorias: esta prueba mide el número de ciclos teniendo exactamente k visitas en una suma acumulativa de un paseo aleatorio.
15. Prueba variante de excursiones aleatorias: esta prueba mide el total de veces que un estado particular es visitado (es decir, se produce) en una suma acumulada de un paseo aleatorio.

Ueli Maurer del Departamento de Ciencia de la Computación de la Universidad de Princeton introdujo este test en 1992 [5]. El test estadístico de Maurer se relaciona estrechamente con la entropía por-bit de la cadena/secuencia, que su autor afirma es "la medida de calidad correcta de una fuente de clave secreta en una aplicación de criptografía". Por lo tanto, la prueba se reclamó para medir la importancia de cifrado real de un defecto, ya que está "relacionada con el tiempo de funcionamiento de una estrategia óptima de búsqueda de la clave", o el tamaño efectivo de la clave de un sistema de cifrado.

El test no está diseñado para detectar un patrón muy específico o el tipo de defecto estadístico. Sin embargo, el test es diseñado "para ser capaz de detectar cualquiera de los defectos estadísticos de clase muy general que pueden ser modelados por una fuente ergódica estacionaria con memoria finita." Debido a esto, Maurer afirma que el test recoge una serie de tests estadísticos estándar.

El test es una prueba de compresión-tipo "basada en la idea de Ziv que un test estadístico universal, se puede basar en un algoritmo de codificación de fuente universal. Un generador debe pasar el test si y sólo si su secuencia de salida no se puede comprimir de forma significativa". Según Maurer, el algoritmo de codificación de fuente debido a Lempel-Ziv "parece ser menos adecuado para su aplicación como un test estadístico", ya que parece ser difícil de definir un test estadístico, cuya distribución puede ser determinada o aproximada.

El test requiere una larga secuencia de bits (en el orden de $10*2^L$ $+1000*2^L$ con $6 \leq L \leq 16$) que es dividida en dos tramos de bloques L-bits ($6 \leq L \leq 16$), Q ($\geq 10*2^L$) bloques de inicialización y K ($\approx 1000*2^L$) bloques de test. Nosotros tomamos $\left\lceil \frac{n}{L} \right\rceil - Q$ para maximizar este valor. El orden de magnitud de Q debe ser elegido específicamente para asegurar que todos los posibles patrones binarios L-bits de hecho se producen dentro de los bloques de inicialización. El test no es adecuado para valores muy grandes de L, porque la inicialización toma tiempo exponencial de la L.

El test vuelve atrás a través de toda la secuencia, mientras que avanza a través del segmento de test de los bloques de L-bits, comprobando la coincidencia exacta anterior más cercana de plantilla L-bit y copiando/guardando la distancia - en número de bloques - a la coincidencia anterior. El algoritmo calcula el $\log_2$ de todo como las distancias para todas las plantillas L-bit en el segmento de test (que da, efectivamente, el número de dígitos en la expansión binaria de cada distancia). A continuación, los promedios de todas las longitudes de expansión por el número de bloques de test.

$$f_n = \frac{1}{K}\left[\sum_{i=Q+1}^{Q+K} \log_2(\#\text{índices desde la anterior aparición de la misma plantilla})\right]$$

El algoritmo realiza esto de manera eficiente por los subíndices de una dinámica tabla de búsqueda haciendo uso de los enteros representación de los bits binarios que constituyen los bloques de la plantilla. Una versión estandarizada de la estadística – la estandarización de ser prescritas por el test - se compara con un rango aceptable basado en un estándar normal (de Gauss) la densidad, haciendo uso de la media de la prueba estadística, que viene dada por la fórmula:

$$Ef_n = 2^{-L}\sum_{i=1}^{\infty}(1-2^{-L})^{i-L}\log_2 i$$

El valor esperado del test estadístico f_n es el de la variable aleatoria $\log_2 G$ donde $G=G_L$ es una variable aleatoria geométrica con el parámetro $1-2^{-L}$.

Además de los métodos estadísticos que hemos visto en el presente capítulo, se dispone de una serie de test que verifican correctamente los generadores de números aleatorios uniformes. Entre los más importantes están, el test DIEHARD [6] y el TestU01 [7]. El TestU01 contiene todos los test realizados por DIEHARD y NIST. Estos paquetes cubren una amplia variedad de test que permiten verificar la aleatoriedad de un juego de muestras aleatorias.

Para la realización de cálculos y pruebas de los test se utiliza la herramienta matemática MatLab que ofrece un entorno computacional, integrado y técnico que combina computación numérica, gráficos avanzados y visualización, y un lenguaje de programación de alto nivel. MatLab incluye funciones para análisis de datos y visualización; números y símbolos de computación; gráficos de ingeniería y ciencia; modelado, simulación y prototipado; y programación, desarrollo de aplicación y diseño de interfaz de usuario grafica.

3. Desarrollo.

En este capítulo se concentran las bases teóricas, cálculos y estudios previos necesarios para la posterior implementación hardware del Circuito de seguridad en el arranque y el Test estadístico de NIST.

La FPGA que se va a utilizar para el desarrollo es una SPARTAN 6 modelo XC6SLX9-3FTG256 de XILINX. Este tipo de FPGA ofrece un bajo consumo y coste, a la vez que un alto rendimiento.

3.1. Circuito de seguridad en el arranque.

El objetivo de esta parte del proyecto es usar las herramientas básicas y componentes de las FPGAs de Xilinx para realizar un circuito automático que sea capaz de individualizar el diseño al que esté acoplado.

¿Cómo?, mediante un sistema de autocomprobación que impida el correcto funcionamiento de la aplicación en caso de clonación.

A continuación se muestra un diagrama de bloques que muestra, simplificado, el funcionamiento de la aplicación de seguridad.

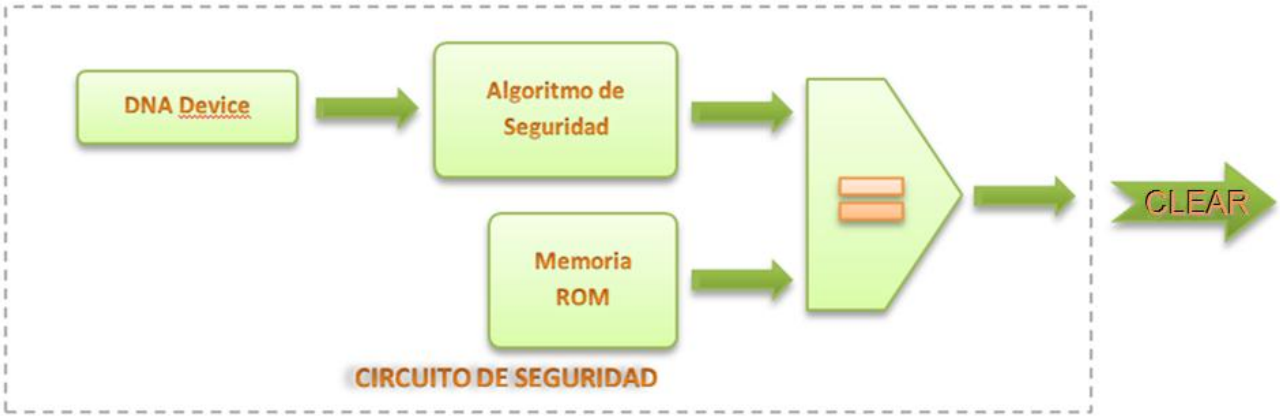


Ilustración 1. Diagrama de bloques del circuito de seguridad

Para conseguir la auto comprobación se siguen una serie de pasos que se clasifican en dos etapas distintas: la configuración externa de la FPGA y la comprobación interna que realiza automáticamente el circuito.

Durante la configuración externa de la FPGA:

- El primer paso se realiza manualmente y consiste en extraer de manera externa el valor DNA identificativo de cada FPGA.
- Después se genera con el uso de una plantilla y mediante un algoritmo de encriptación, otro valor que se guardará en una memoria ROM del circuito. Este valor será el que se utilice como valor de comprobación en el proceso de comparación.
- Para individualizar cada diseño, se ha de modificar el circuito para cada dispositivo FPGA introduciendo el valor de la memoria ROM. Realizar este cambio en el diseño puede ser un proceso de larga duración dependiendo de la complicación del circuito que se desee proteger ya que el tiempo de implementación es directamente proporcional a la complejidad del circuito. Por este motivo, el sistema de modificación se hará en plano, es decir, se realizarán los cambios sobre el fichero binario base de configuración de la FPGA (bitstream), sin la necesidad de re-implementación.

Una vez reconfigurado el circuito se descarga el bitstream en la FPGA y el dispositivo está listo para ser utilizado. Internamente, se realiza la comprobación automática en el arranque de la FPGA:

- El circuito lee internamente el valor del DNA y lo encripta con el mismo algoritmo que se usa exteriormente.
- El último paso es la comparación de ambos valores. La señal de salida CLEAR es la encargada de controlar el resto del circuito, si la comprobación es correcta esta señal habilitará el resto de procesos, en el caso contrario la señal de CLEAR inhabilitará el funcionamiento.

Los posteriores apartados corresponden al estudio necesario para la obtención del DNA, la memoria ROM y la reconfiguración del circuito para cada valor de DNA.

3.1.1. DNA.

El dispositivo DNA es un identificador único de 57 bits introducido en la FPGA durante el proceso de manufactura realizado por los fabricantes, en este caso el fabricante del dispositivo utilizado es Xilinx.

Es un valor de lectura únicamente, al que se puede acceder *externamente* a través del puerto JTAG mediante la interfaz IMPACT de ISE, *o internamente* por el componente DNA_PORT, ilustración 2, utilizándolo en un diseño mediante el lenguaje de descripción de circuitos VHDL.

Este valor de 57 bits se caracteriza por tener dos partes, los 55 bits menos significativos (0,…,54) son únicos para cada FPGA y el bit 56 y 55 son fijos a '1' y '0' respectivamente, ver la ilustración 3. El DNA se lee en serie siendo el bit más significativo ('1') el primero en ser leído.

Para acceder al valor DNA internamente se ha utilizado el componente integrado DNA_PORT. Este componente es sencillamente un registro de desplazamiento con una entrada serie DIN y una salida serie DOUT. Para que el registro de desplazamiento DNA_PORT comience su funcionamiento debe estar activa la señal READ durante un flanco de subida de CLK, a continuación se activa la señal SHIFT que debe estar a nivel alto durante toda la salida de bits que se produce en cada flanco de subida del reloj CLK.

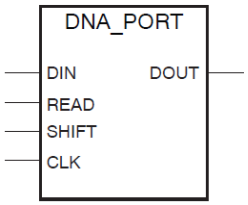


Ilustración 2. Bloque DNA_PORT

A continuación se explican todos los puertos del componente, su funcionamiento y modo de conexión:

- DIN: entrada serie. Esta entrada puede estar conectada de 3 formas distintas:
  - DIN conectada a '0': DOUT es el valor DNA de 57 bits.
  - DIN conectada a una cadena de bits: DOUT es el valor DNA concatenado a dicha cadena.
  - DIN conectada a DOUT: DOUT es el valor DNA mostrado cíclicamente.
- READ: activa el comienzo de la lectura con un pulso a nivel alto.
- SHIFT: se mantiene a nivel alto durante la lectura del DNA.
- CLK: señal de reloj.

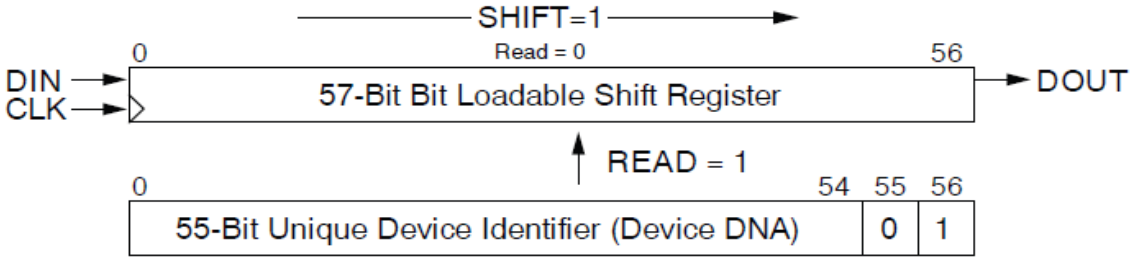


Ilustración 3. Descripción esquemática de la primitiva DNA_PORT

Para comprobar el funcionamiento de esta instancia que ofrece Xilinx, se diseña un circuito de control cuya entrada es READ, se genera la señal SHIFT y se muestra el resultado serie de DOUT. La entrada DIN del DNA_PORT estará conectada a un '0' lógico.

La señal de reloj del circuito se genera mediante la instancia STARTUP_SPARTAN6 que genera una señal oscilatoria de 50 Mhz y se modifica dicha frecuencia con el Digital Clock Manager DCM_CLKGEN para obtener una frecuencia menor de 2 Mhz, necesaria para la lectura del DNA.

Existe una valor predeterminado para la simulación del DNA ("000000000000000"h) o la opción de simular cualquier otro valor modificando el campo SIM_DNA_VALUE => X"130E92FB87D22AC" del generic map de la instancia en VHDL [8]. El resultado de la simulación es el siguiente para el SIM_DNA_VALUE antes descrito:

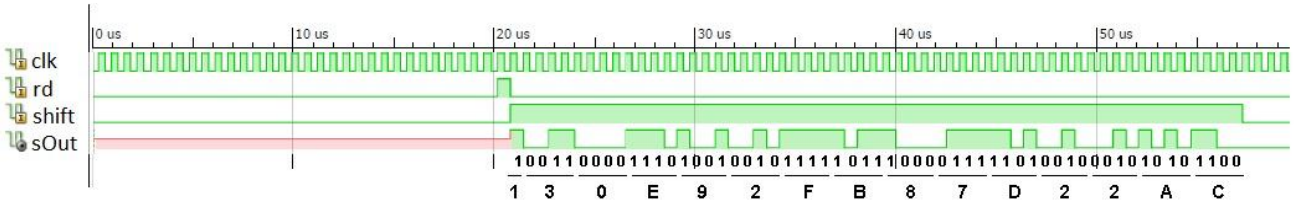


Ilustración 4. Simulación de funcionamiento del DNA_PORT

Con la simulación del circuito de obtención de DNA se comprueba el funcionamiento interno del DNA_PORT para poder trabajar con él en la implementación del circuito de seguridad.

Para la obtención del valor real del DNA de una determinada FPGA se conecta el dispositivo al PC mediante el puerto JTAG y se accede a la herramienta IMPACT de Xilinx ISE en modo consola (batch mode). Después de configurar el dispositivo e identificarlo, se accede al valor de DNA mediante el comando *readdna –p <pos>, [*9].

Durante las pruebas de laboratorio de usará una FPGA Spartan 6 cuyo DNA es el mostrado a continuación y se obtiene con una serie de comandos sencillos descritos en el fichero *rddna.cmd* descrito en el Anexo I.

DNA: "100110000111010010010111110111000011111010010001010101100" en binario o "130E92FB87D22AC" en hexadecimal.

Con estas pruebas se puede dar por finalizado el estudio del comportamiento del DNA y su método de obtención correcto consiguiendo así las bases de la primera etapa del circuito o aplicación: obtención externa e interna del identificador DNA.

3.1.2. Memoria ROM de 16 bits.

Para el siguiente paso en el desarrollo del circuito se debe crear una memoria ROM fija en unas determinadas celdas físicas. Esta premisa es necesaria para poder encontrar

posteriormente la relación entre los datos que se quieran guardar en dicha memoria ROM y los datos binarios del bitstream obtenido de la síntesis del circuito.

Para facilitar la búsqueda de relaciones entre los distintos ficheros el estudio se realizará en base a una memoria más pequeña de 16 bits. Estudiando las distintas posibilidades que ofrece el lenguaje VHDL y las librerías de ISE Xilinx, se decide utilizar un registro de desplazamiento con la posibilidad de establecer un valor de inicio de hasta 16 bits (componente SRL16E de las librerías de VHDL). Este valor de inicio es un campo genérico del componente, INIT=> X"13f5", donde la palabra de 16 bits se especifica con cuatro dígitos hexadecimales.

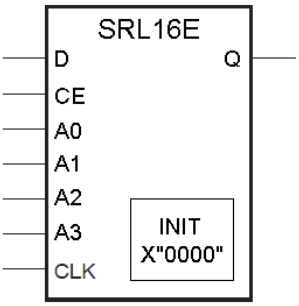


Ilustración 5. Bloque SRL16E.

Conectando las 4 entradas A0,.., A3 a '1' se consigue que el registro de desplazamiento sea de 2^4 = 16 bits pudiendo usar el valor de inicio como memoria. La salida cambia en el flanco de subida de CLK si la señal de habilitación CE esté activa a nivel alto. La salida se conecta de nuevo a la entrada para que sea cíclico.

Se comprueba que se puede utilizar este componente y su valor de inicio para formar una memoria de lectura con salida serie, lo cual es apropiado para nuestro propósito ya que la salida del DNA también es en serie.

Limitaciones.

Una vez realizado un pequeño circuito en el que se puede leer una salida serie de una memoria de 16 bits creada a partir del registro de desplazamiento, se deben fijar las celdas físicas.

La manera más intuitiva de realizar este proceso es usando la herramienta PlanAhead [10] de Xilinx a la que se accede desde la ventana de implementación del circuito. Esta herramienta es capaz de gestionar proyectos desde los ficheros VHDL al bitstream, con una interfaz de usuario fácil e intuitiva. Permite ver los resultados de implementación, de tiempo, distribución de pines, restricciones, etc.

Desde esta herramienta se puede fijar los puertos o pines (NETs) o componentes (Primitives) del circuito en la posición que se hayan establecido automáticamente al

sintetizar el diseño, o modificarlas para volver a sintetizar y poder acortar o mejorar los retardos de las señales.

Como se observa en la siguiente imagen hay que seleccionar el componente o puerto en la ventana superior de la izquierda y automáticamente aparecen sus propiedades en la ventana inferior de la izquierda con la posibilidad de fijar la instancia en unas celdas determinadas, descritas por sus coordenadas físicas (p.e. X34Y29), activando la opción *Fixed*. De esta forma este componente siempre ocupará una posición estática en posteriores cambios y sintetizaciones del diseño.

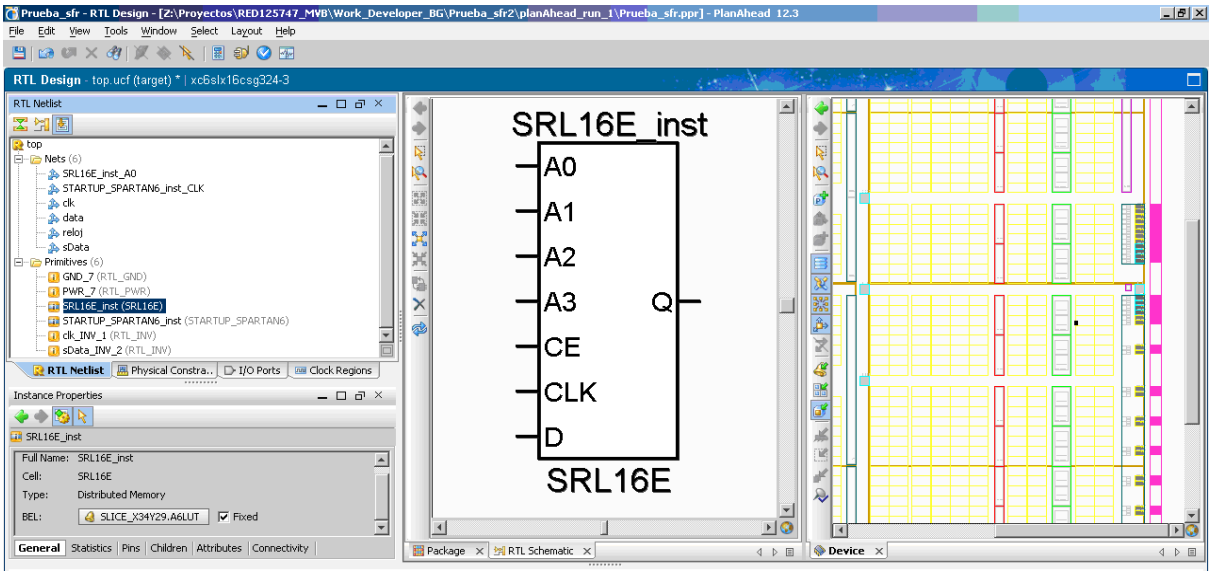


Ilustración 6. Vista de PlanAhead del componente SRL16E

Ampliando la vista de la ventana derecha de la imagen anterior vemos físicamente la celda **SLICE_X34Y29.A6LUT** (resaltada en azul) y un ejemplo del resto de distintos componentes que se pueden usar en el tipo SLICEM como se muestra en la ilustración 7.

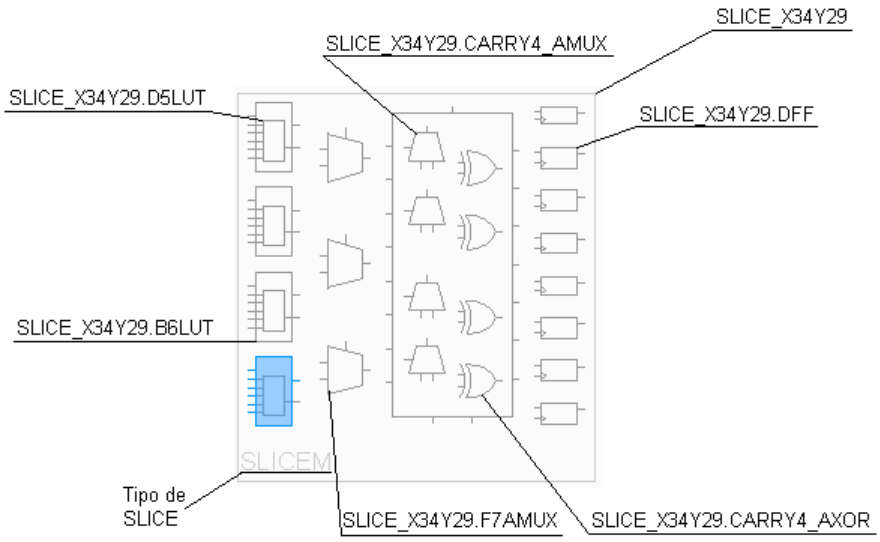


Ilustración 7. SLICE de una FPGA.

Aplicando la misma metodología fijamos el resto de puertos e instancias para realizar un circuito lo más estático posible. El fichero de extensión *.ucf (constraints file)* es el resultado en modo texto de las restricciones realizadas gráficamente con la herramienta PlanAhead y se genera automáticamente. Este archivo contiene las siguientes líneas descriptoras:

```
#rom16bits.ucf
NET "data" LOC = F15;
INST "STARTUP_SPARTAN6_inst" LOC = STARTUP;
INST "SRL16E_inst" BEL = A6LUT;
INST "SRL16E_inst" LOC = SLICE_X34Y29;
```

A partir de este ejemplo se observa la relación del lenguaje descriptivo y el tipo de instancia que se quiere fijar. Esta información se usará para la creación del fichero *.ucf* del diseño final ya que es un método de configuración menos intuitivo pero más rápido que el ofrecido por la herramienta PlanAhead.

3.1.3. Búsqueda de patrones.

Este estudio está orientado a encontrar un método de modificación del fichero binario base de configuración de la FPGA, que contiene la implementación del circuito. Esta modificación solo afecta al valor de memoria ROM que se utiliza como parámetro de comparación, y es la forma en la que se individualiza cada circuito para una única FPGA.

En esta etapa del proyecto se utiliza la herramienta libre *VBindiff* que es capaz de comparar y editar dos archivos binarios.

En la imagen que se muestra a continuación se enseña un ejemplo de la vista que ofrece el programa y de cómo se nombrará en la documentación posterior la **posición** de una pareja de datos en hexadecimal, por su columna y fila.

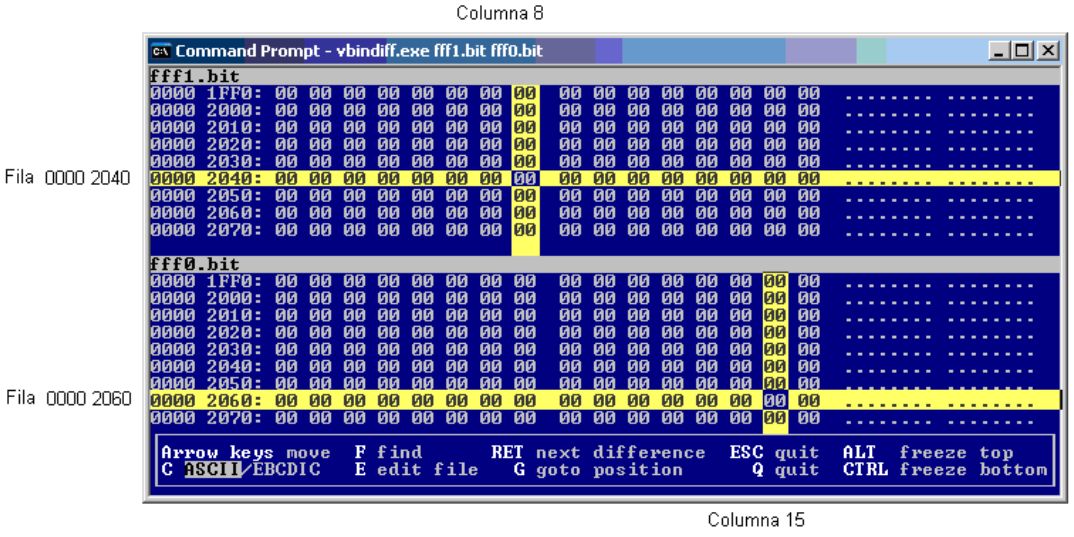


Ilustración 8.Interfaz de VBinDiff.

Comparación de Bitstreams con el mismo valor de Memoria ROM.

La primera prueba que se hace es la de comparar el mismo circuito, sin cambios pero sintetizado varias veces para poder comprobar si existen diferencias circunstanciales y si realmente ha tenido éxito la técnica de fijar las instancias físicamente en la FPGA. Para ello también se sintetiza un diseño obviando el fichero de restricciones (rom16bit.ucf), es decir, dejando el diseño sin restricciones sobre las celdas físicas.

Los resultados son claros en la comparación de estos ficheros, siempre cambia una parte del bitstream aunque el diseño no se modifique. Los cambios que son independientes del fichero son los referentes a la hora y fecha en la que se realiza la síntesis del circuito:

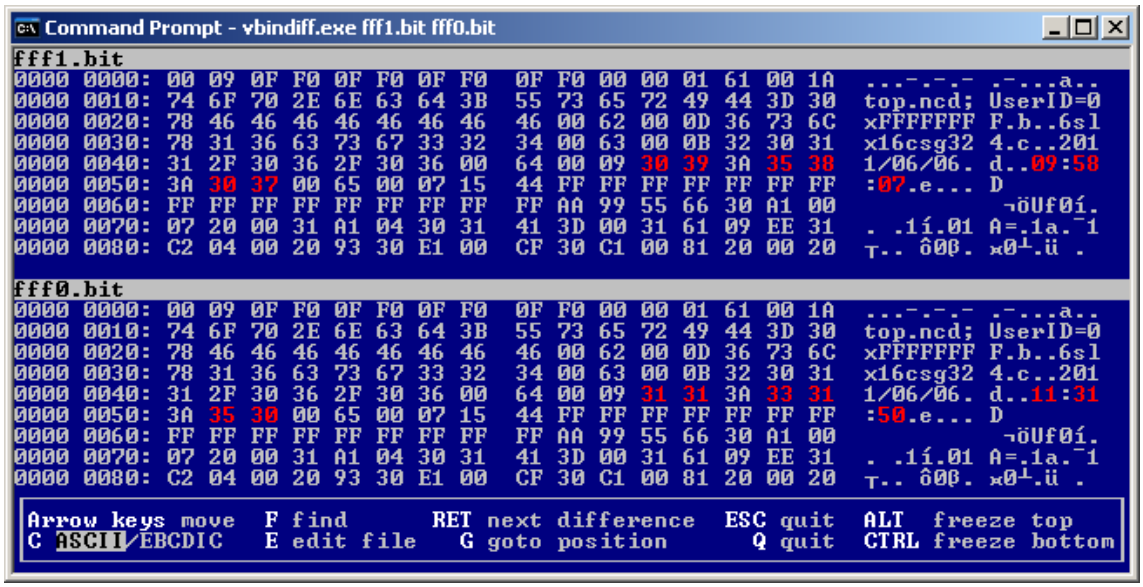


Ilustración 9. Modificaciones temporales del bitstream.

Estos datos siempre están ubicados en las mismas posiciones dentro del bitstream, lo cual es una buena señal para creer que sucederá lo mismo en el caso de los relativos a la memoria ROM.

También se observan unos cambios independientes del diseño que se producen siempre en las mismas posiciones del fichero binario y al final de este:

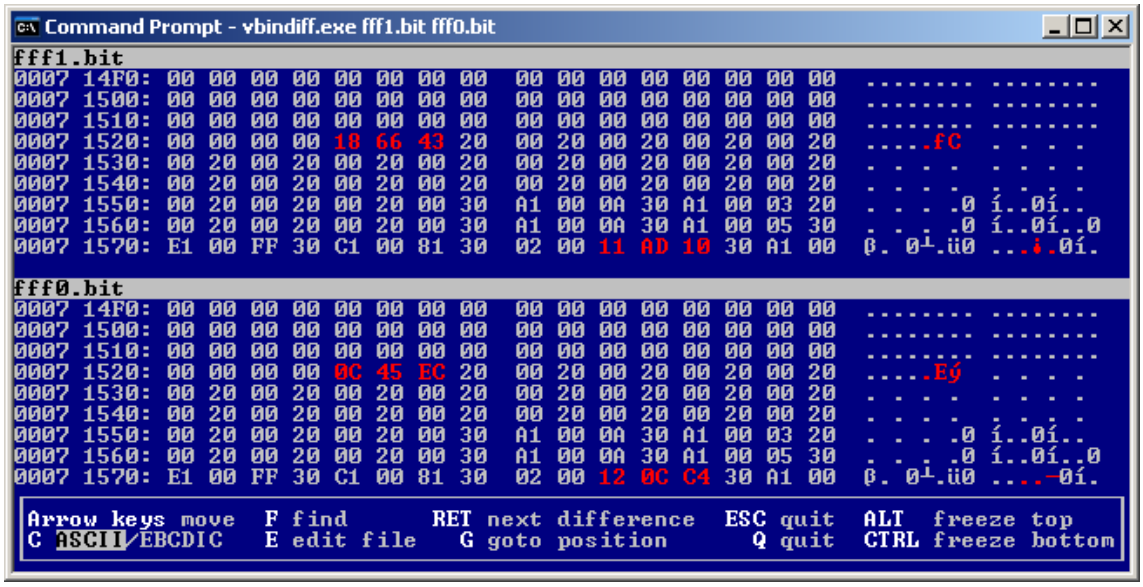


Ilustración 10. Modificaciones independientes del bitstream.

No se ha averiguado su significado, pero estos y los cambios temporales serán obviados como cambios significativos para el posterior estudio.

Comparación de Bitstreams con distinto valor de Memoria ROM.

Para comparar bitstreams con distintos valores de Memoria ROM se modifica en el generic map de SRL16E el campo INIT de configuración del valor inicial del componente. Este campo descrito en apartados anteriores, es un valor de 16 bits en hexadecimal, por esta razón y para facilitar el manejo de los bitstreams, cada nuevo fichero binario sintetizado será nombrado con el valor en hexadecimal de la memoria ROM.

A continuación se muestra una tabla con los resultados de las diferencias de los distintos bitstreams:

Fila	Col.	0001.BIT	5BC7.BIT	DEC1.BIT	FFF1.BIT	5551.BIT	FFF0.BIT	FF00.BIT	FF10.BIT	FF20.BIT	FFD0.BIT	0002.BIT	5552.BIT	FFF2.BIT	FFF3.BIT	FFF4.BIT	FFF5.BIT	FFF6.BIT	FFF7.BIT	FFF8.BIT	FFF9.BIT	FFFB.BIT	FFFF.BIT	5555.BIT	DEC4.BIT
0002 D1A0	14	00	CC	0C	0F	00	0F	00	00	03	0C	C0	C0	CF	CF	0F	0F	CF	CF	3F	3F	FF	FF	00	0C
0002 D1A0	15	00	F0	FC	FF	00	FF	FF	FF	FF	FF	00	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	00	FC
0002 D220	16	C0	FC	CC	CF	CF	0F	00	03	00	0F	00	0F	0F	CF	3F	FF	3F	FF	0F	CF	CF	FF	FF	3C
0002 D230	1	00	CF	3F	FF	FF	FF	FF	FF	FF	FF	00	FF	FF	FF	FF	FF	FF	FF	FF	FF	CF	FF	FF	3F

Tabla 1. Pruebas en base a distintos valores de ROM

Se observa que hay cambios en 4 pares de datos hexadecimales del binario, por lo que en el siguiente apartado se buscará la relación entre las parejas de datos del binario y el valor en hexadecimal de la memoria ROM.

Máscaras para la búsqueda del patrón

Si la memoria ROM es del valor “ABCD” el binario es nombrado ABCD.bit siendo “A” el dígito menos significativo.

La primera comparación se centrará la atención en buscar los cambios relativos al dígito más significativo, en el caso del ejemplo anterior ABC**D**.bit.

Fila	Col.	0000.BIT	0001.BIT	0002.BIT	0003.BIT	0004.BIT
0002 D1A0	**14**	00	00	**C**0	**C**0	**0**0
0002 D1A0	**15**	00	00	00	00	00
0002 D220	**16**	00	**C**0	00	**C**0	**3**0
0002 D230	**1**	00	00	00	00	00

Tabla 2. Resultados de modificar el dígito más significativo “D”.

Existe una relación directa entre este dígito de la memoria y un par de dígitos hexadecimales que cambian en el fichero binario. Para encontrar la relación del resto de dígitos de la memoria repetimos el proceso cambiando los valores en el SRL16E.

A continuación se muestran los resultados en base al segundo dígito más significativo, el AB**C**D.bit.:

Fila	Col.	0000.BIT	0010.BIT	0020.BIT	0030.BIT	0040.BIT
0002 D1A0	**14**	00	00	0**3**	0**3**	00
0002 D1A0	**15**	00	00	00	00	00
0002 D220	**16**	00	0**3**	00	0**3**	0**C**
0002 D230	**1**	00	00	00	00	00

Tabla 3. Resultados de modificar el segundo bit más significativo "C".

Se itera con el resto de dígitos de la ROM para todos los valores posibles en hexadecimal. Al repetir el proceso y con las primeras pruebas en las que se eligieron valores aleatorios se llega a la conclusión de que cada dígito $\mathbf{D_n}$ tiene su imagen en el binario en forma de dos dígitos $\mathbf{d_{n1}}$ y $\mathbf{d_{n2}}$, siendo **n** un valor del 0 al 3 que indica el peso de cada dígito, de menos a más significativo.

Fila	Col.	$D_0D_1D_2D_3$.BIT
0002 D1A0	**14**	d_{31} d_{21}
0002 D1A0	**15**	d_{11} d_{01}
0002 D220	**16**	d_{32} d_{22}
0002 D230	**1**	d_{12} d_{02}

Tabla 4. Relación entre los dígitos de la Memoria ROM y los del bitstream.

Además de la relación entre estos valores se observa que la información está codificada. En un principio se buscó como patrón una máscara de bits, pero después de varios análisis se observó que existían dos tipos de codificaciones que se han denominado A y B.

La codificación tipo A la usan los dígitos $\mathbf{D_0}$ y $\mathbf{D_2}$.

$\mathbf{d_{n1}}$	0	0	3	3	0	0	3	3	C	C	F	F	C	C	F	F
Dato ROM D_0 o D_2	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
$\mathbf{d_{n2}}$	0	3	0	3	C	F	C	F	0	3	0	3	C	F	C	F

Tabla 5. Codificación tipo A.

La codificación tipo B Este se usa en los dígitos D_1 y D_3.

d_{n1}	0	0	C	C	0	0	C	C	3	3	F	F	3	3	F	F
Dato ROM D_1 o D_3	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
d_{n2}	0	C	0	C	3	F	3	F	0	C	0	C	3	F	3	F

Tabla 6. Codificación tipo B.

Para comprobar los resultados se hace una última prueba y se comparan los resultados teóricos con los prácticos.

Si en la memoria ROM se establece el valor **"5BC7"** en hexadecimal el bitstream tendría los siguientes datos según las codificaciones anteriores:

Fila	Col.	5BC7.BIT
0002 D1A0	**14**	C C
0002 D1A0	**15**	F 0
0002 D220	**16**	F C
0002 D230	**1**	C F

Tabla 7. Ejemplo numérico de relación entre la memoria ROM y el bitstream.

El resultado de la síntesis del circuito con dicho valor de ROM genera un bitstream de la siguiente forma:

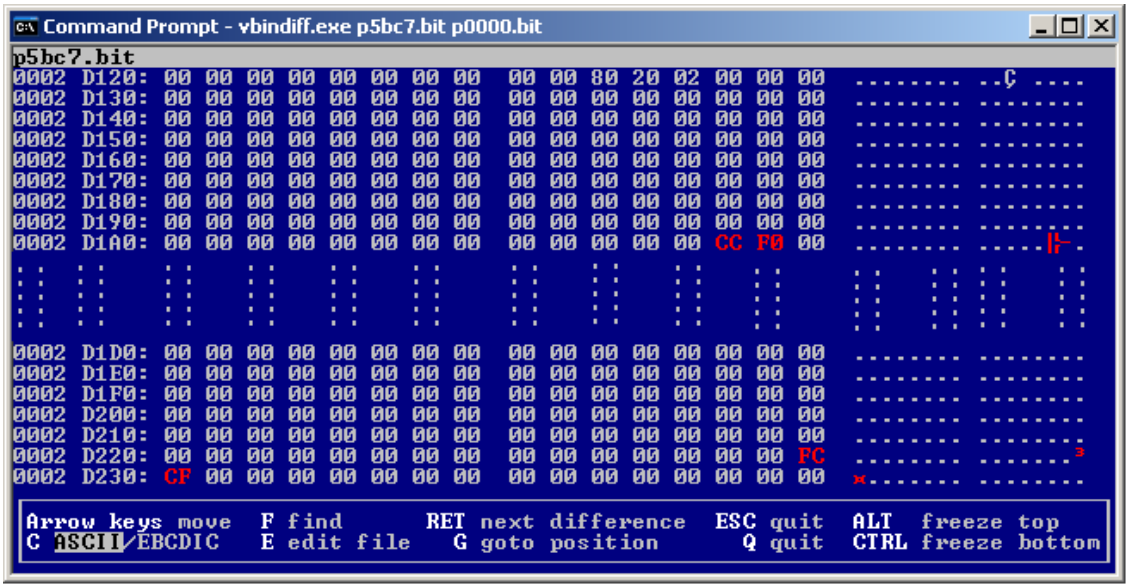


Ilustración 11. Resultados en VBinDiff para el ejemplo numérico.

Los datos previstos son correctos y se finaliza con éxito el estudio previo para el circuito de seguridad.

3.2. Test estadístico de Maurer.

Uno de los tests estadísticos del NIST de verificación de aleatoriedad de RGNs y PRGNs es el Test Universal de Maurer.

El algoritmo criptográfico que se desea verificar posee una longitud de datos de salida adaptable y se elige la óptima para las características del test implementado. La señal de CLEAR que proviene del circuito de seguridad se utiliza como señal de inicialización del circuito de test. Y la salida final del circuito es la señal Aleatoria_Test que indica si la muestra analizada es aleatoria o no.

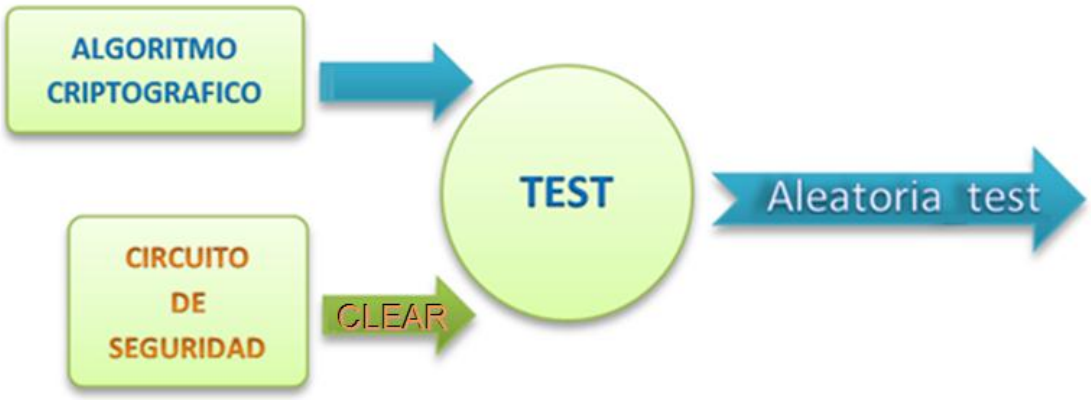


Ilustración 12. Diagrama de bloques del test estadístico.

3.2.1. Principios básicos y funcionamiento.

En el caso del Test Universal de Maurer existe una relación directa entre el *P-value* y la función de error complementario, *erfc().*

$$erfc(z) = \frac{2}{\sqrt{\pi}} \int_z^{\infty} e^{-u^2} du \qquad \textbf{(0)}$$

En base al Capítulo 2 sobre las condiciones de aleatoriedad y bajo la conclusión de que la secuencia es aleatoria, se puede calcular el valor de *x* que cumple:

$$P - value = erfc(x) > 0.001 \qquad \textbf{(1)}$$

$$x < erfc^{-1}(0.001) = 2.3267 \qquad \textbf{(2)}$$

Test Maurer´s "Universal Statistical".

Una vez explicadas los conceptos estadísticos generales, es necesario profundizar en la teoría del Test de Maurer. El objetivo de esta prueba es detectar si una secuencia de bits se repite con mucha frecuencia. Para ello se analiza la muestra que se divide en bloques de una longitud determinada y se busca la última repetición de cada bloque.

A continuación se muestran una serie de definiciones necesarias para la comprensión del test:

n	Número de bits a analizar. Se especifican distintos rangos de números de bits de la muestra para definir los parámetros utilizados en el test. En este diseño se elige el mínimo valor de muestra posible para garantizar el análisis estadístico correcto, $n = 387840$.
L	Número de bits de cada bloque en el que se divide *n.* Para el valor *n* elegido, $L = 6$.
Q	Número de bloques de la etapa de inicialización, $Q = 10 \times 2^L = 640$.
K	Número de bloques de la etapa de análisis, $K = 1000 \times 2^L = 64000$.
i	Índice que señala el bloque de *L* bits que se está analizando en un momento determinado del test, $i = \{1,2,3...,(Q+K)\}$.
j	Representación decimal del contenido de cada bloque de *L* bits, $j = \{0,1,2,3...,63\}$.
T_j	Tabla que contiene la última ocurrencia de cada bloque de *L* bits, $T_j = i$. La dimensión de la tabla *Tj* corresponde con el máximo valor decimal que se puede representar con *L* bits, $\dim(T) = 2^L = 64$.
Sum	La suma del logaritmo en base 2 de la distancia entre dos bloques iguales. $$Sum = \sum_{i=Q+1}^{Q+K} \log_2(i - T_j) \quad \textbf{(3)}$$
f_n	La suma del $\log_2$ de la distancia entre dos bloques iguales en relación al número de bloques analizados. $$f_n = \frac{Sum}{K} \quad \textbf{(4)}$$

La secuencia a estudiar de *n* bits se subdivide en *Q+K* bloques de *L* bits. El estudio se divide en dos etapas:

- Etapa de inicialización: en esta etapa se estudian los bloques denotados por *i en 1 a Q.* Para cada *i* de completa la tabla T_j.
- Etapa de análisis: etapa para analizar los bloques denotados por *i en Q+1 a K+Q.* En esta etapa se realiza el cálculo de *Sum* para cada *i*, y se vuelve a completar la tabla T_j.

Para obtener una información más detallada ver el capítulo 2.9 del paquete de test estadísticos de NIST [11].

Una vez adquirido el sumatorio total de todos los bloques a analizar, se calcula la relación con el *P-value* y la *erfc.* Partiendo de la ecuación 2:

$$x = \left| \frac{f_n - \exp ectedValue(L)}{\sqrt{2}\sigma} \right| < 2.3267 \quad \textbf{(5)}$$

$$f_n < 2.3267 \cdot \sqrt{2} \cdot \sigma + \exp ectedValue(L) \quad \textbf{(6)}$$

$$f_n > -2.3267 \cdot \sqrt{2} \cdot \sigma + \exp ectedValue(L) \quad \textbf{(7)}$$

Todos los parámetros necesarios para la obtención de fn se calculan en base a las definiciones anteriores de *L* y *K,* y los siguientes parámetros y ecuaciones*:*

$$\exp ectedValue(L) = \exp ectedValue(6) = 5.2177052$$

$$Varience(L) = Varience(6) = 2.954$$

$$c = 0.7 - \frac{0.8}{L} + (4 + \frac{32}{L}) \frac{K^{-3/L}}{15} \quad \textbf{(8)}$$

$$\sigma = c\sqrt{\frac{\text{var}\, ience(L)}{K}} \quad \textbf{(9)}$$

Utilizando la ecuaciones 4 hasta la 9 se llega a los resultados finales de:

$$Sum_{\min} = f_{n\min} \cdot K = 3.327376654976987\text{e} + 005$$

$$Sum_{\max} = f_{n\max} \cdot K = 3.351292145023014\text{e} + 005$$

La conclusión de que la secuencia sea aleatoria será si el valor se *Sum* está comprendido entre los límites establecidos por Sum_{min} *y* Sum_{miax}. Todos los cálculos se han realizado mediante la herramienta matemática MatLab para obtener la mayor fiabilidad posible (ver Anexo I).

3.2.2. Consideraciones y cálculo de umbrales para el test.

Con el fin de simplificar se elige una aproximación de un decimal para realizar los cálculos matemáticos. Es decir, todos los valores son considerados 10 veces mayores a los originales y se redondean a la cifra entera, ya que las FPGA trabajan con señales sin decimales.

Los límites superior e inferior del valor *Sum* quedan establecidos en:

$$S_{\min} = 3327377 \quad y \quad S_{\max} = 3351292 \qquad \textbf{(10)}$$

Otro de los factores restrictivos que observamos para poder realizar la posterior implementación del test de seguridad, es el cálculo de logaritmos en base 2. Este tipo de operación es complicada para la implementación hardware, por ello la opción que se elige como solución es generar una memoria que guarde dichos valores.

Según las bases teóricas del test, los parámetros dimensionales que se han elegido y la ecuación 3, se llega a la conclusión de que es necesario obtener el logaritmo en base 2 de valores comprendidos entre el 1 y el 64640 (*Q+K)* para el peor de los casos (que el último bloque a analizar nunca se hubiese repetido durante el análisis, *i=64640 y* T_j*=0*). Aunque este caso es bastante improbable ya que sólo existen 64 valores posibles para cada bloque, debe estar incluido para garantizar la eficacia de la prueba.

Las siguientes gráficas muestran el logaritmo en base 2 de valores comprendidos entre 1 y 64640 calculados con MatLab. La imagen de la izquierda muestra el $\log_2$ con 16 decimales, mientras que la gráfica de la derecha son valores aproximados a un decimal, 10 veces mayores y redondeados a la cifra entera.

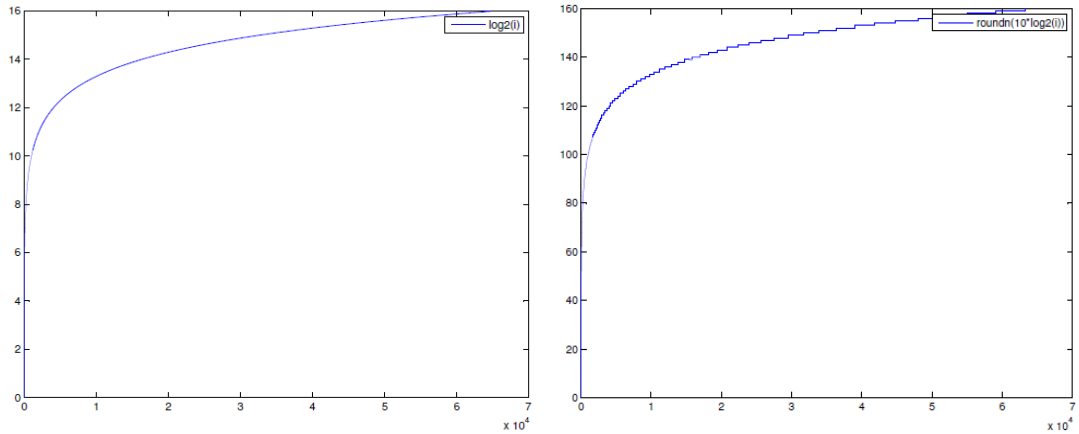


Ilustración 13. Gráficas comparativas de $\log_2$ de 1 a 64640 (MatLab).

Se observa visualmente que la aproximación de un decimal para el diseño es bastante acertada, y se puede usar perfectamente sin que afecte al resultado final de *Sum.*

En la tabla 8 se muestran los valores máximos y mínimos de los logaritmos 10 veces mayores y redondeados a la cifra entera:

i	1	64640
Parte entera de $10x\log_2(i)$	0	160

Tabla 8. Valores máximo y mínimo de $\log_2$

Una memoria de estas características supone una gran cantidad de recursos. Exactamente ocupa 64640 palabras de 8 bits necesarios para representar números naturales hasta 160, es decir, una capacidad aproximada de 505Kbits. Al tratarse logaritmos, la diferencia entre resultados consecutivos disminuye a la vez que *i* crece y al redondear a las décimas se observa que se repiten valores y que se puede generar una memoria más reducida.

Con la ayuda de MatLab se calculan todos los valores de los logaritmos y se guardan en una memoria de 64.640 valores. Mediante un algoritmo se realiza un bucle para generar dos memorias de menor tamaño, una que contenga la posición donde se empieza a repetir un valor de logaritmo (POS) y otra con el valor del logaritmo repetido (CONT). Dicho proceso se ejemplifica a continuación, en la tabla 9, con valores supuestos para facilitar la comprensión:

i	**Log(i)**
1	10
2	20
3	30
4	40
5	40
6	40
7	50
8	50
9	50
10	50
11	50
12	60
13	60

→

POS	**CONT**	**n**
1	10	1
2	20	2
3	30	3
4	40	4
7	50	5
12	60	6

Tabla 9. Proceso de reducción de memoria.

Al realizar esta modificación se consigue finalmente dos memorias de *n* palabras. El valor definitivo de *n* es 136:

- El contenido de POS es un valor comprendido entre 1 y 64640, por lo que son necesarios 16 bits para representarlo. La capacidad de esta memoria es de 136x16bits, aproximadamente 2Kbits.
- El contenido de CONT es el valor de logaritmo, igual que el de la memoria inicial, que se representa con 8bits. Esta memoria es de 136x8bits, poco más de 1Kbits.

En este paso del desarrollo hemos conseguido reducir una memoria de 505Kbits por dos memorias que suman como máximo 3,2Kbits. Desde la aplicación de MatLab se pueden realizar todos los procesos necesarios y generar dos archivos ASCII con los valores de POS y CONT para su posterior utilización en el circuito (ver Anexo I).

Como método de análisis del error cometido al utilizar valores logarítmicos con 1 decimal se calcula el error absoluto y relativo obtenido en el test estadístico sobre varias muestras aleatorias. Para ello se emplea MatLab como herramienta de cálculo:

- Paso 1. Se realiza una base de datos (vector) de todos los valores de $\log_2$ del 1 al 64640 (*Q+K)* con un formato que abarca hasta 16 cifras decimales: vector *x.*
- Paso 2. Se crea otro vector de los valores anteriores redondeados a las décimas: vector *w.*
- Paso 3. Se realiza el test estadístico de Maurer basado en la ecuación 3 cuyos valores utilizados de logaritmo están pre calculados y guardados en los vectores *x (16 decimales)* y *w* (1 decimal).
- Paso 4. El error absoluto, *Ea,* será la diferencia entre los resultados finales del sumatorio para las distintas aproximaciones de los valores de logaritmos.
- Paso 5.Para calcular el error relativo, *Er,* se toma como referencia de valor real el de mayor exactitud, es decir, el calculado con números de 16 decimales.

En la siguiente tabla 10 se muestran ejemplos de este proceso de cálculo para distintas muestras aleatorias:

	Muestra 1	**Muestra 2**	**Muestra 3**	**Muestra 4**
***Sum* (16 decimales)**	334308	333370	334427	333732
***Sum* (1 decimal)**	334376	333436	334500	333807
Error absoluto	68	66	73	75
Error relativo	$2{,}03\text{x}10^{-4}$	$1{,}97\text{x}10^{-4}$	$2{,}18\text{x}10^{-4}$	$2{,}25\text{x}10^{-4}$
Error relativo (%)	0,02	0,02	0,02	0,02

Tabla 10. Errores absolutos y relativos de cálculos.

Se puede dar por válido el criterio elegido de redondeo a un decimal ya que como muestran los resultados de la tabla 10, el error relativo medio (0,02 %) es mínimo y aceptable para el desarrollo del test.

4. Implementación hardware.

En este capítulo se implementa el circuito completo basado en el estudio realizado durante el Capítulo Desarrollo del proyecto. Al igual que el apartado anterior, se diferencia en dos partes: el circuito de seguridad y el test estadístico.

En este capítulo no se incluyen los ficheros descritos en VHDL, para consultarlos ver el Anexo II.

4.1. Implementación del circuito automático de seguridad en el arranque.

El funcionamiento de este circuito es el siguiente: usando el valor identificativo DNA se pretende crear un circuito de comprobación con un valor guardado en una memoria ROM de 64 bits. Si los valores coinciden el circuito funcionará con normalidad, en caso contrario, se activará una señal de CLEAR que inhabilite el sistema completo.

El diseño se divide en 6 componentes distintos que se describirán en los siguientes apartados. Es un sistema automático por lo que no tiene puertos de entrada; tiene una salida llamada CLEAR que indica si la comparación es correcta o no y una salida llamada VIDA que muestra si el reloj interno funciona o no, es decir, una señal de vida.

RESET	Señal de entrada asíncrona que inicializa todos los registros y señales.
CLEAR	Señal de salida Inicialmente a ‘0’; Después de la comprobación será ‘1’ si ésta es correcta, si no ‘0’
VIDA	Señal de salida de vida. Señal periódica para el parpadeo de un LED

El diagrama de bloques del circuito es el siguiente:

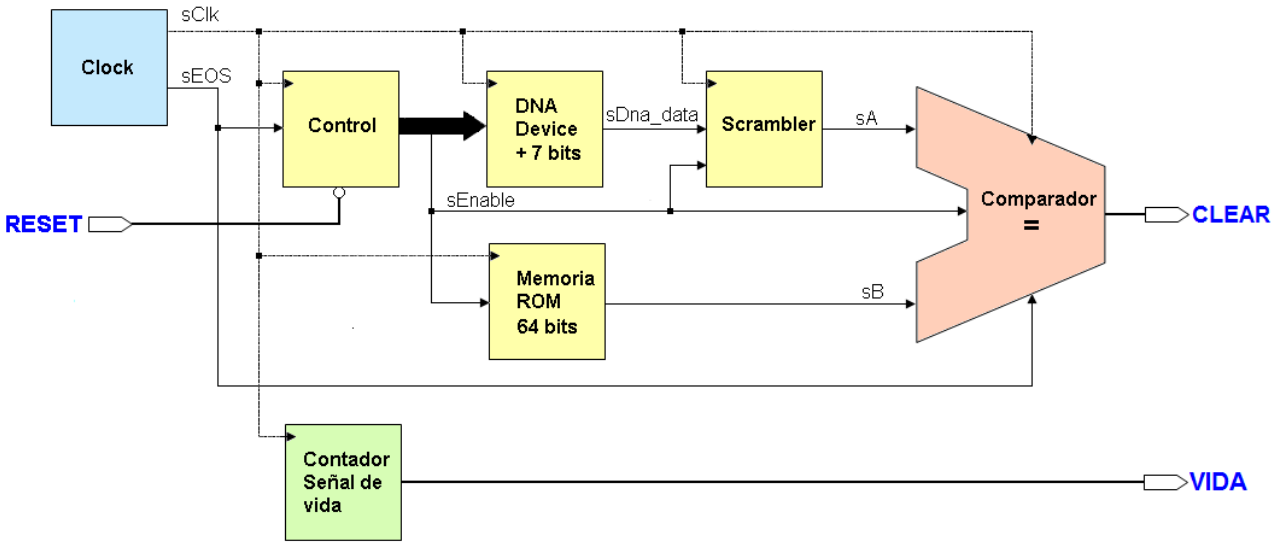


Ilustración 14. Diagrama de bloques esquemático del circuito de seguridad.

4.1.1. Componentes.

Reloj interno – CLOCK.

Con el componente STARTUP del dispositivo SPARTAN 6 se genera una señal periódica de reloj de 50MHz cuya frecuencia se modifica con el administrador de reloj digital. Con este componente denominado DCM se realiza un divisor de la señal por 32 y se obtiene una frecuencia de 1.5625MHz y un periodo de la señal de reloj CLOCK de 640ns. La frecuencia de reloj máxima para obtener el valor DNA es de 2 MHz [12].

La salida EOS del componente STARTUP que indica si se ha realizado satisfactoriamente la configuración del reloj se usa como señal de inicialización de otras señales y componentes del circuito completo.

Las señales de entrada, salida o internas del componente se resumen en la siguiente tabla:

CLOCK	Señal de salida de reloj del circuito de frecuencia 1.5625MHz. Generada internamente.
ES	Señal procedente del STARTUP que indica si éste se ha configurado correctamente.

La señal interna sClk50 es una señal de 50 Hz procedente del STARTUP Y sClk2 es la señal de salida de reloj antes del búfer del reloj.

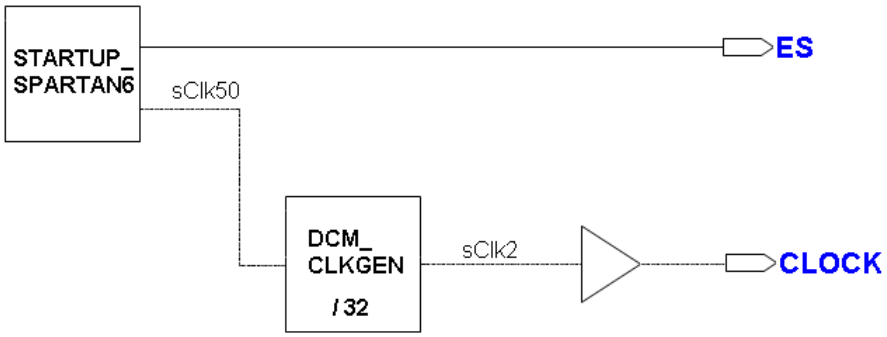


Ilustración 15. Circuito del RELOJ interno.

La siguiente imagen muestra el periodo de la señal de reloj que es 640ns:

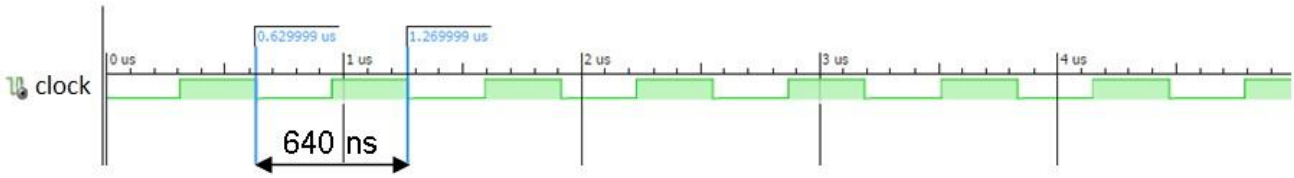


Ilustración 16.Simulación del RELOJ interno.

Para comprobar que el reloj funciona correctamente existe el Contador Señal de vida, que genera una señal periódica de 1,3 s aproximadamente y que se saca por un puerto que está conectado a un LED el cual parpadea el reloj ha sido configurado con éxito. No existe la posibilidad de simular el funcionamiento de la señal ES.

Circuito de control – CONTROL.

El circuito de control es el encargado de generar las señales necesarias para el funcionamiento del DNA_PORT y otras señales de sincronización entre componentes del circuito. Este componente está formado principalmente por un contador de ciclos de reloj, un generador de un pulso y una máquina de estados.

Las señales de salida y entrada del componente son:

CLK	Señal de entrada de reloj de 1,5625MHz. Generada internamente.
RESET	Señal de entrada asíncrona que inicializa todos los registros y señales.
ES	Señal de entrada que indica la configuración correcta del reloj. Se usa como señal de inicialización.
LEER	Señal de salida de un pulso de duración a nivel alto generada para activar

	la entrada READ del DNA_PORT.
C56	Señal de salida activa a nivel alto durante 56 ciclos de reloj después del flanco de bajada de LEER, generada para activar la entrada SHIFT del DNA_PORT .
C64	Señal de salida activa a nivel alto durante 64 ciclos de reloj después del flanco de bajada de LEER. Es la señal de enable del circuito completo para poder sincronizar los componentes.

Otras señales internas del circuito son:

Aux2	Señal std_logic_vector (6 downto 0) que se usa para contar el número de ciclos de reloj.
Temp	Señal de inicialización del contador de de ciclos de reloj.
CE	Chip enable del generador de un pulso.

El diagrama de bloques del circuito es:

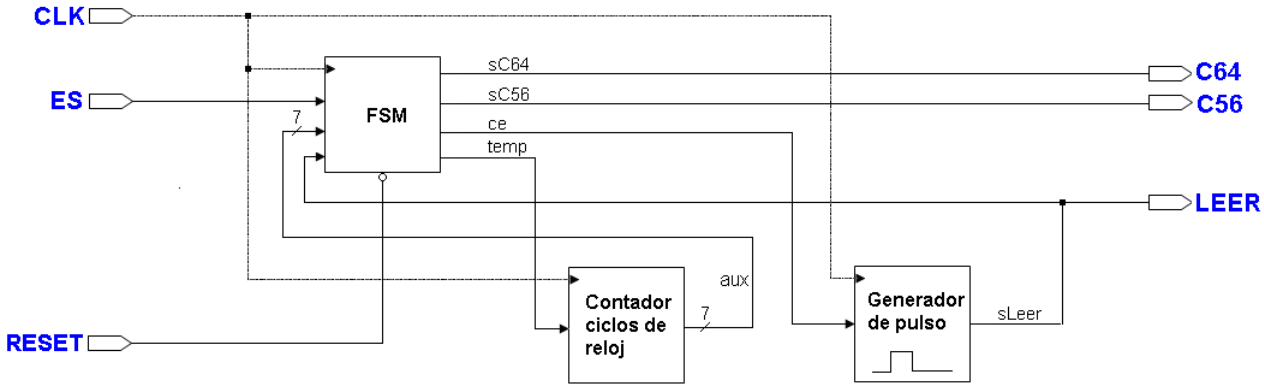


Ilustración 17. Circuito de CONTROL.

La máquina de estados del circuito se mantiene en el estado INICIO hasta que la señal ES indica que la configuración del reloj es correcta, y todas las señales de salida están en un estado de inicialización. Una vez que la configuración del reloj ha finalizado, se pasa al estado de LECTURA y se habilita el CE del generador de pulso LEER. Inmediatamente después de que LEER esté a nivel alto se prosigue con el estado SHIFT que habilita la señal del contador de pulsos y genera las salidas SC56 y SC64, que cuentan 56 y 63 pulsos respectivamente. En este estado se realiza la cuenta hasta 56 ciclos de reloj, y en el siguiente, ENABLE, se continúa hasta llegar a 63 pulsos. El estado FINAL es un estado de reposo. El diagrama de estados se muestra a continuación y las salidas son CE, Temp, sC56 y sC64:

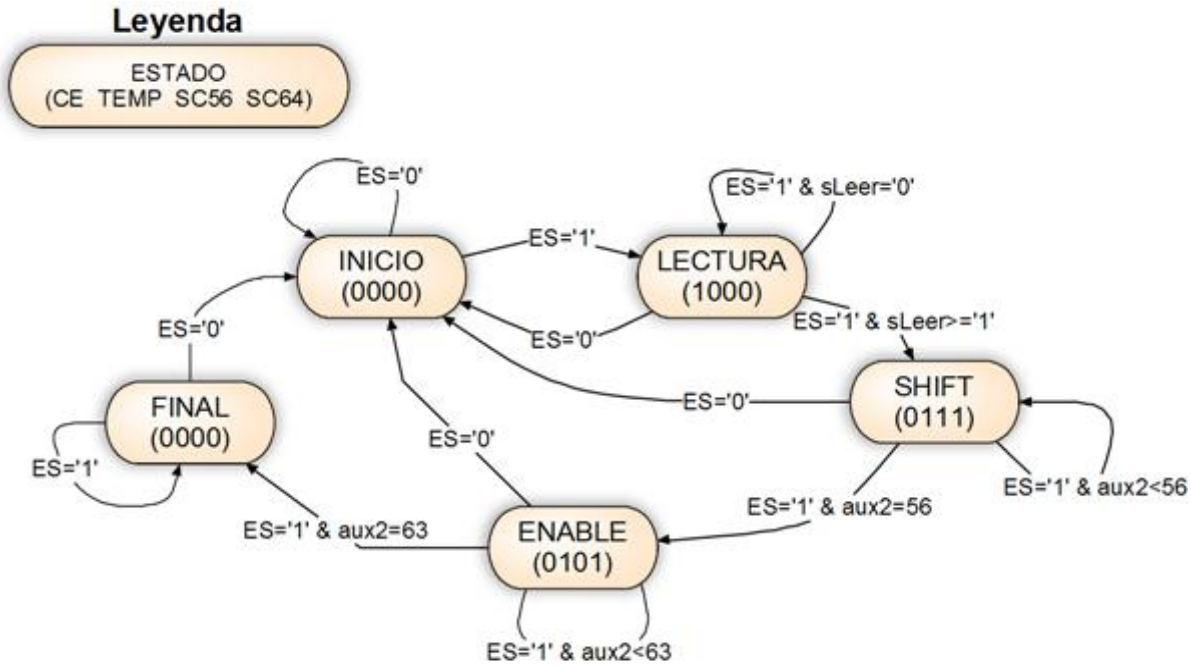


Ilustración 18. Máquina de estados del CONTROL.

En la simulación observamos el comportamiento de todas las señales antes mencionadas excepto Aux2, encargada de contar los pulsos de reloj, ya que no es apreciable su cambio. Se ha señalizado con marcadores el tiempo en el que las señales C56 y C64 están a nivel alto.

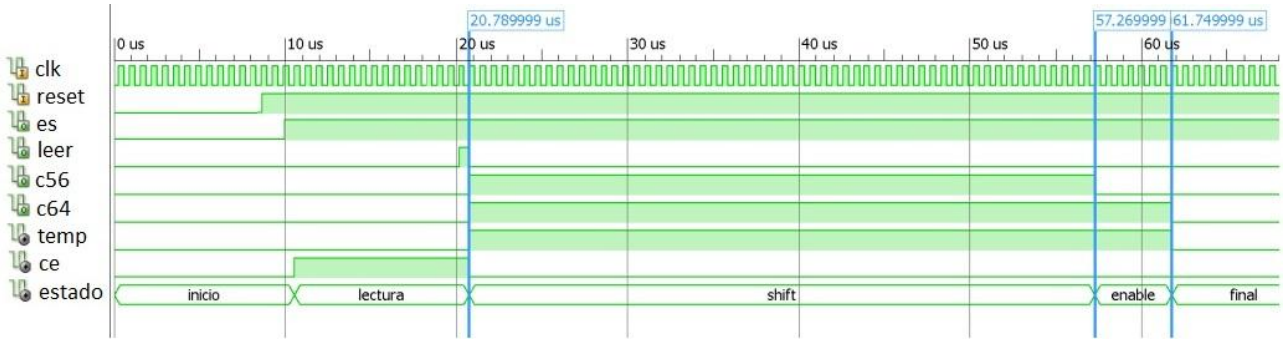


Ilustración 19. Simulación de CONTROL.

Circuito de lectura del DNA – DNA.

Para completar la cadena de 57 bits procedentes del DNA se añaden 7 bits en serie con un registro de desplazamiento. En concreto la cadena de 7 bits que se añade es fija y de la forma “1111000”.

Para su funcionamiento necesita las señales propias de entrada del DNA_PORT descritas en el capitulo 3.1.1 y la señal de sincronización.

CLK	Señal de reloj de 1.5625MHz. Generada internamente
RD	Señal de un pulso de duración a nivel alto para activar la entrada READ del DNA_PORT.
SHIFT	Señal activa a nivel alto durante 56 ciclos de reloj después del flanco de bajada de RD, generada para activar la entrada SHIFT del DNA_PORT .
ENABLE	Señal activa a nivel alto durante 64 ciclos de reloj después del flanco de bajada de RD. Es la señal de enable del registro de desplazamiento para desplazar los 7 bits de inicio y los 57 del DNA.
DNA_DATA	Señal de salida compuesta por 7 bits + 57 bits del DNA Device.
sOUT	Salida DOUT del DNA_PORT.

Cada una de las señales de salida del registro de desplazamiento se denomina Qn y se conectan en serie.

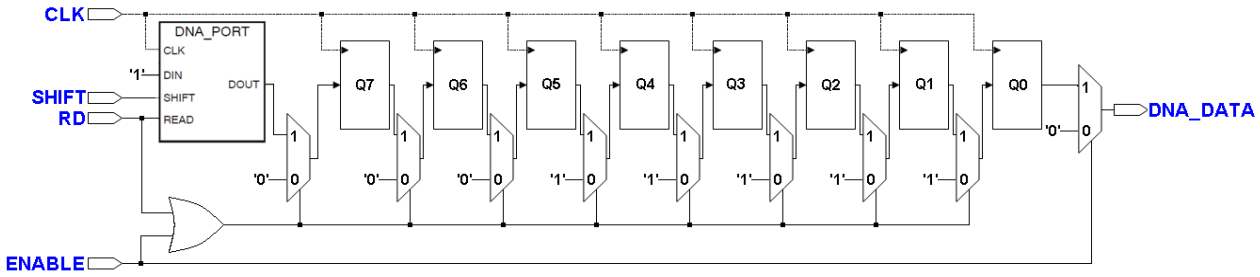


Ilustración 20. Circuito de DNA.

La primera marca señala el final de los 7 primeros bits desplazados e independientes del DNA. La segunda marca muestra el final de la trama de 64 bits.

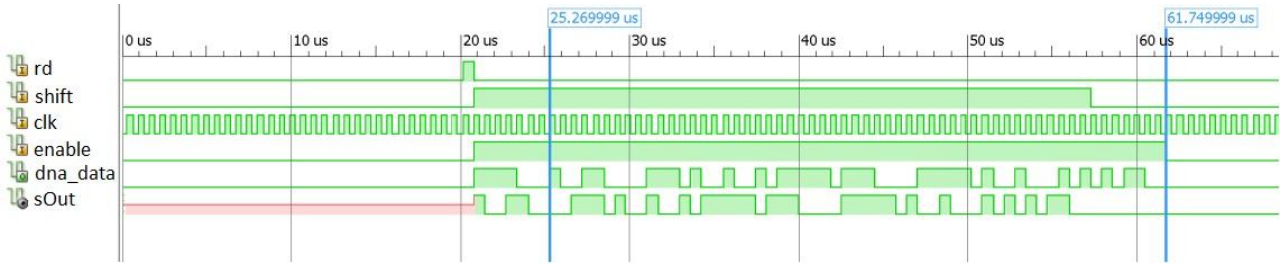


Ilustración 21. Simulación de DNA.

Memoria Rom de 64 bits – ROM.

La memoria ROM de 64 bits se forma con las especificaciones de la memoria de 16 bits probada en el estudio previo.

Se compone de 2 registros de desplazamiento de 32 bits conectados en cascada que existen en las librerías de componentes de Xilinx.

Lo más importante a destacar de este componente es el campo INIT de los registros SRL32CE_0 y SRL32CE_1. La siguiente tabla muestra un ejemplo de la configuración de este campo con la información de la memoria de 64 bits expresada en 16 dígitos hexadecimal de la forma $\mathbf{D_n}$, siendo **n** el índice que indica el peso de cada dígito de menos a más significativo.

	SRL32CE_1	**SRL32CE_0**
INIT =>	X"8D13C2CE"	X"A1222A66"
D_n	X" $D_0D_1D_2\ldots D_7$"	X" $D_8D_9D_{10}\ldots D_{15}$"

Tabla 11. Relación entre Dn e INIT.

Para configurar los SRL en su máxima capacidad existe la entrada A de 5 bits que se conecta al valor "11111". Este componente posee también dos salidas distintas, una de ellas específica para conectarlo en cascada llamada Q31.

Las puertas de entrada, salida o señales internas se describen a continuación:

CLK Señal de reloj de 1,5625MHz.

ENABLE Señal activa a nivel alto durante 64 ciclos de reloj. Es la señal de enable para sincronizar la salida del SRL de la memoria ROM con el resto del circuito.

ROM_DATA Señal de salida de la memoria ROM de 64 bits .

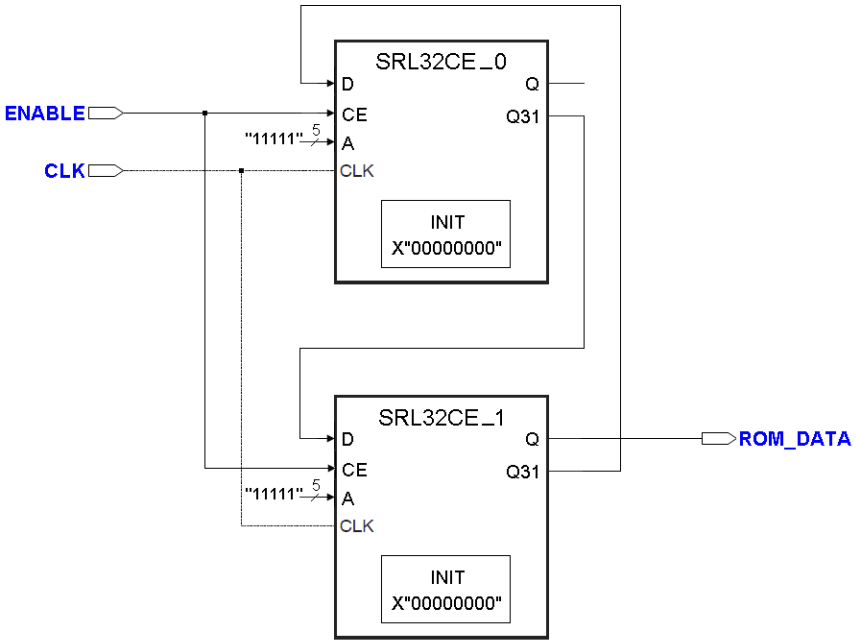


Ilustración 22. Circuito de la memoria ROM.

Los resultados de la simulación de una memoria ROM de valor en hexadecimal "8D13C2CEA1222A66" son:

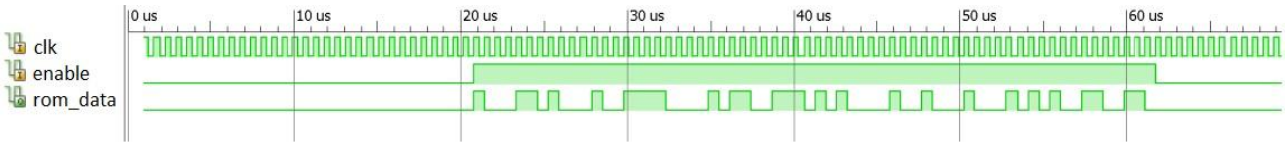


Ilustración 23. Simulación de ROM.

Circuito de encriptación – SCRAM.

El algoritmo de seguridad que se utiliza es un polinomio semejante a un CRC. Las propiedades de CRC están definidas por la longitud del polinomio generador y sus coeficientes. Para implementarlo se diseña un registro de desplazamiento con unos valores iniciales y la señal sScr es una señal de realimentación del circuito.

CLK	Señal de entrada de reloj de 1,5625MHz. Generada internamente
ENABLE	Señal de entrada activa a nivel alto durante 64 ciclos de reloj. Es la señal de enable para sincronizar la salida de SCRAM con el resto del circuito.
DATAIN	Es la entrada al algoritmo procedente del componente DNA. Su longitud es de 64 bits.
DATAOUT	Señal de salida de 64 bits en serie con el valor de DNA modificado.

Con este algoritmo de encriptación se consiguen 21 estados cíclicos distintos que se muestran a continuación:

Estado	Q1	Q2	Q3	Q4	Q5	sScr
0	1	0	1	0	0	0
1	0	1	0	1	0	1
2	1	0	1	0	1	1
4	1	1	0	1	0	1
5	1	1	1	0	1	1
6	1	1	1	1	0	0
7	0	1	1	1	1	0
8	0	0	1	1	1	0
9	0	0	0	1	1	0
10	0	0	0	0	1	1
11	1	0	0	0	0	0
12	0	1	0	0	0	0
13	0	0	1	0	0	0
14	0	0	0	1	0	1
15	1	0	0	0	1	1
16	1	1	0	0	0	0
17	0	1	1	0	0	0
18	0	0	1	1	0	1
19	1	0	0	1	1	0
20	0	1	0	0	1	1
0	1	0	1	0	0	0

Tabla 12. Estados de SCRAM.

El diagrama de bloques del componente es:

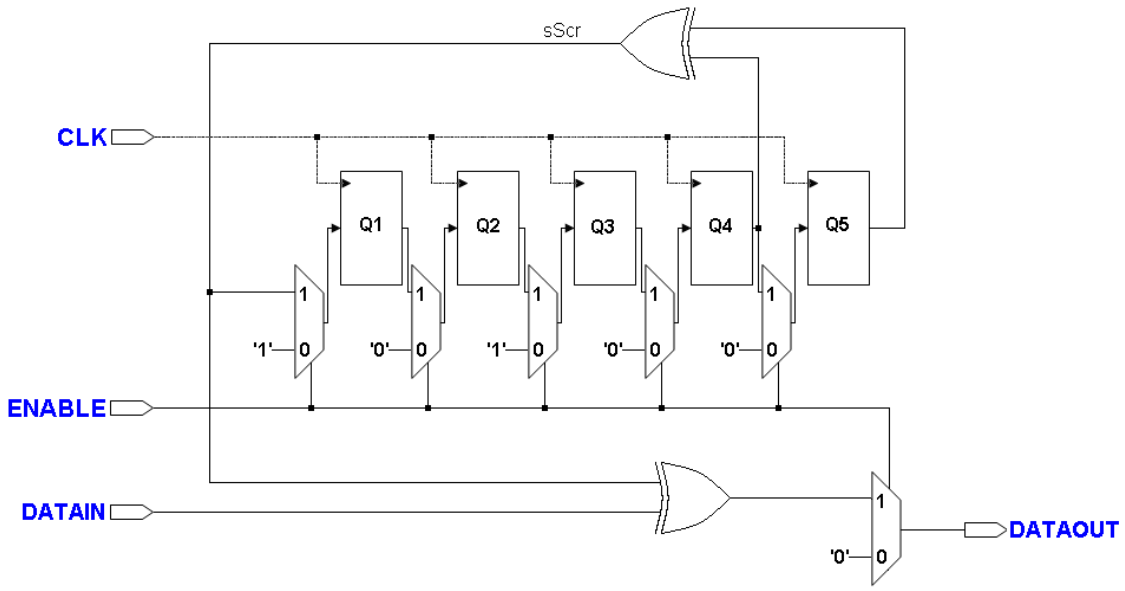


Ilustración 24. Circuito de SCRAM.

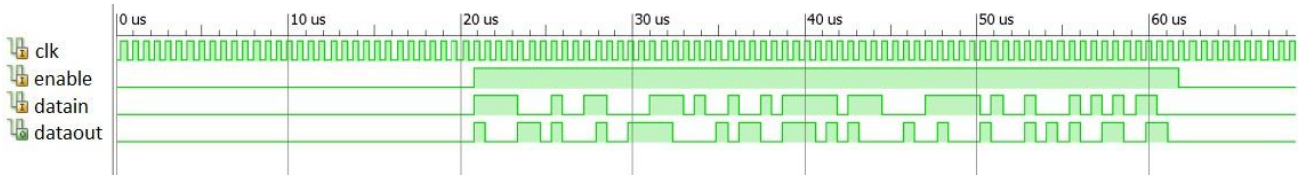


Ilustración 25. Simulación de SCRAM.

Circuito comparador de señales – COMPARE.

Este es el paso final de la aplicación donde se comparan las salidas del componente SCRAM con la memoria ROM. Si el resultado de esta comparación es correcto la salida CLEAR se mantendrá a '1', si por el contrario los bits no coinciden esta señal será '0' a partir del primer cambio que encuentre.

La comparación se realiza bit a bit en cada ciclo de reloj durante el estado alto de la señal ENABLE. Las señales de

CLK	Señal de entrada de reloj de 1.5625MHz. Generada internamente
ENABLE	Señal de entrada activa a nivel alto durante 64 ciclos de reloj. Es la señal de enable para sincronizar el periodo de comparación.
ES	Señal de entrada procedente del componente CLOCK que indica si este está correctamente configurado. Se utiliza como señal de inicialización.
A	Señal de entrada procedente del SCRAM que contiene el valor de DNA modificado.
B	Señal de entrada procedente de la memoria ROM.
CLEAR	Señal de salida resultado de la comparación. Si A=B CLEAR='1', si no '0'. Esta señal será igual que sComp cuando ENABLE sea '0' y Aux1 sea '1'.

Otras señales internas de interés son:

Aux0	Señal para realizar un detector de flanco de subida de ENABLE.
Aux1	Señal que se mantiene a nivel alto una vez comenzado el análisis de comparación entre A y B. Sirve para delimitar el periodo a partir del cual se puede leer la salida CLEAR correctamente. La comparación no ha finalizado si esta señal es '0' o la señal ENABLE es '1'.
sComp	Señal que se activa a un nivel alto al inicio de la comparación de A y B. Si A es distinto de B cambia a un nivel lógico '0'.

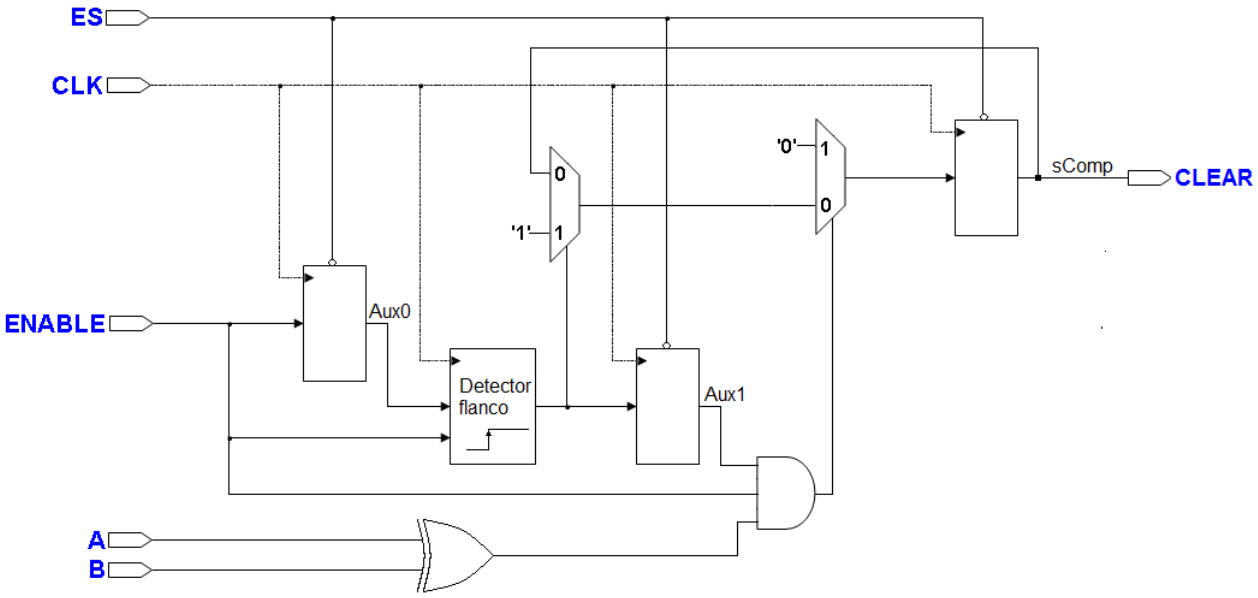


Ilustración 26. Circuito de COMPARE.

Esta es la simulación para el caso en que La memoria ROM esté configurada para esta única FPGA, es decir, la comparación es correcta.

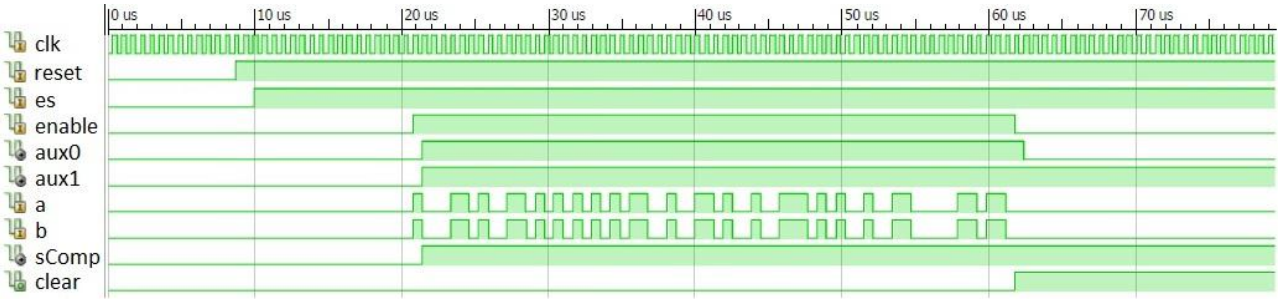


Ilustración 27. Simulación correcta de COMPARE.

Si se clonara el circuito y se descargara en otra FPGA, el DNA cambiaría y por tanto no coincidirían los valores. La siguiente simulación muestra el caso de plagio.

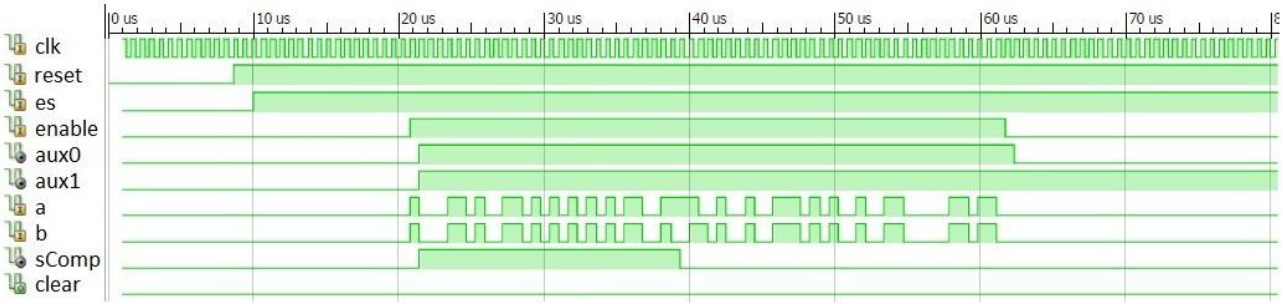


Ilustración 28. Simulación incorrecta de COMPARE.

Las marcas señalan la inicialización de CLEAR y el instante en que cambia porque A es distinto de B.

4.1.2. Relación de Señales y Puertos.

La siguiente tabla facilita la relación entre las señales internas y los puertos componentes que forman el circuito. En los apartados anteriores se describen dichas señales y el uso que cada componente da de ellas.

SEÑAL	MAIN	CLOCK	CNTROL	DNA	ROM	SCRAM	COMPARE
sClk	-	CLOCK	CLK	CLK	CLK	CLK	CLK
sEOS	-	ES	ES	-	-	-	ES
sLeer	-	-	LEER	RD	-	-	-
sShift	-	-	C56	SHIFT	-	-	-
sEnable	-	-	C64	ENABLE	ENABLE	ENABLE	ENABLE
sA	-	-	-	-	-	DATAOUT	A
sB	-	-	-	-	ROM_DATA	-	B
sDna	-	-	-	DNA_DATA	-	DATAIN	-
sCLEAR	CLEAR	-	-	-	-	-	CLEAR
sAux(20)	VIDA	-	-	-	-	-	-

Tabla 13. Relación entre señales y puertos.

4.1.3. Restricciones. Archivo *.ucf.

Las restricciones físicas del diseño son las siguientes:

```
#main.ucf
INST "*/STARTUP_SPARTAN6_inst" LOC = STARTUP;

NET "Clr" LOC = F15;
NET "VIDA" LOC = G16;

INST "*/SRLC32E_0" BEL = A6LUT;
INST "*/SRLC32E_0" LOC = SLICE_X20Y21;

INST "*/SRLC32E_1" BEL = A6LUT;
INST "*/SRLC32E_1" LOC = SLICE_X20Y20;
```

Las salidas VIDA y RST del circuito se han asociado a los pines F15 y G16 que están conectados a dos LEDs de la tarjeta de pruebas de la FPGA.

Los componentes de tipo SRLC32E se han posicionado en dos SLICE cercanos para evitar retrasos entres sus señales.

4.1.4. Síntesis del circuito de seguridad.

Una vez completado todo el diseño del circuito de seguridad se sintetiza para comprobar los recursos que utiliza de la FPGA y las limitaciones de tiempo. A continuación se muestra una copia de los apartados más importantes del Informe de Síntesis que se genera al realizar la implementación completa del circuito:

```
 Device utilization summary:
---------------------------
Selected Device : 6slx9ftg256-3
Slice Logic Utilization:
 Number of Slice Registers:              46  out of  11440      0%
 Number of Slice LUTs:                   61  out of   5720      1%
    Number used as Logic:                58  out of   5720      1%
    Number used as Memory:                3  out of   1440      0%
       Number used as SRL:                3
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     64
   Number with an unused Flip Flop:      18  out of     64     28%
   Number with an unused LUT:             3  out of     64      4%
   Number of fully used LUT-FF pairs:    43  out of     64     67%
   Number of unique control sets:         4
IO Utilization:
 Number of IOs:                           2
 Number of bonded IOBs:                   2  out of    186      1%
Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:                1  out of     16      6%
```

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 3.451ns (Maximum Frequency: 289.800MHz)
   Minimum input arrival time before clock: 2.078ns
   Maximum output required time after clock: 3.856ns
   Maximum combinational path delay: 0.000ns
```

```
Found area constraint ratio of 100 (+ 5) on block main, actual
ratio is 1.
```

Los resultados reafirman que se trata de un diseño sencillo que no limita el espacio de la FPGA (ratio de 1 sobre 100), ni la velocidad (frecuencia de 289Mhz). Con estas características se puede incluir este circuito como un componente de seguridad para mejorar las prestaciones de cualquier otro diseño, ya sea en este caso, el test estadístico para el análisis de algoritmos criptográficos o en otros casos, para múltiples circuitos diseñados sobre FPGAs Spartan 6.

4.2. Implementación del Test de Maurer.

En este apartado se desarrolla una descripción del circuito necesario para realizar el test Universal de Maurer.

La entrada del circuito es un algoritmo criptográfico o generador de números aleatorios. Para el diseño se ha decidido analizar un PRNG cuya salida de datos tiene una longitud configurable, es decir, se puede determinar el tamaño de los datos y muestra que sea analizan. Según los parámetros seleccionados para este tipo de test, elegimos una configuración tal que se puedan analizar datos de *L=6* bits cada ciclo de reloj, y que se obtengan *n=387840* muestras.

A continuación se explica detalladamente las características de los puertos de entrada y salida:

CLK	Señal de entrada de reloj externa. Debe ser la misma que utilice el PRGN que se desea analizar y de una frecuencia máxima de 50Mhz.
RESET	Señal de entrada asíncrona que inicializa todos los registros y señales.
START	Señal de entrada que inicia y habilita el análisis estadístico cuando esta a nivel alto.
S_N	Entrada de datos a analizar. Tiene una longitud determinada de 6 bits y cambia cada ciclo de reloj.
Aleatoria_test	Señal de salida de 1 bit que indica en su nivel alto si la muestra es aleatoria. Esta señal es válida únicamente si la señal FIN es ‘1’ también.
Value_test	Señal de salida de 23 bits sumatorio final del logaritmo de las diferencias entre la posición de un bloque y su anterior repetición. Corresponde al valor *Sum* de la ecuación 3.
FIN	Señal de salida que indica si el análisis estadístico ha finalizado cuando es igual a ‘1’.

También se utiliza en la implementación del circuito la señal CLEAR procedente del componente SEGURIDAD:

CLEAR	Señal de inicialización de todos los registros y valores utilizados. Si CLEAR es ‘0’ no hay funcionamiento.

El funcionamiento del test requiere un contador que indique que bloque se bits se está analizando, es decir, *i* desde 1 a 64640 (*Q+K).* Dependiendo de la posición de *i*, se pueden diferenciar 4 etapas distintas:

- Etapa 0: Si *i=0;* en este periodo el test aún no ha comenzado porque RESET, CLEAR o START son '0'.
- Etapa 1: Si *i>0* y *i≤Q;* es la etapa de inicialización de la tabla *Tj.* En este periodo se lee el bloque *i* y se escribe en la posición de memoria de datos TABLA correspondiente al valor decimal del contenido del bloque.
- Etapa 2: Si *i>Q* y *i≤Q+K;* es la etapa de análisis. En esta etapa se busca la última repetición del bloque *i* que se está analizando, se calcula la diferencia entre la posición actual y la repetición, y se realiza el sumatorio de logaritmos. Después se completa la TABLA de manera idéntica que en la etapa de inicialización.
- Etapa 3: Si *i>Q+K;* Fin del test, en este periodo la señal FIN se activa, y son válidos los resultados de las señales ALEATORIA_TEST y VALUE_TEST.

A continuación se muestra un diagrama esquemático del circuito:

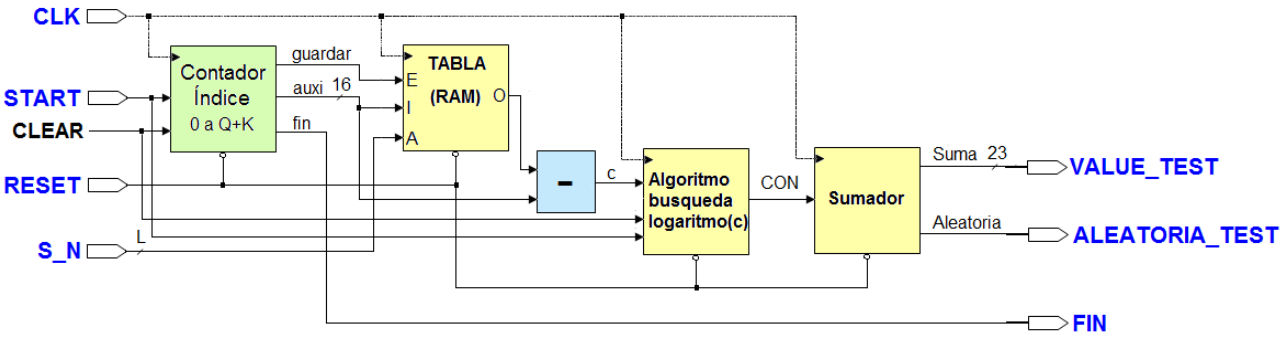


Ilustración 29. Diagrama de bloques del circuito de test estadístico.

Para entender mejor los siguientes apartados ver el capítulo 3.2 donde se explican las bases teóricas y funcionamiento del test.

Para entender mejor los siguientes apartados ver el capítulo 3.2 donde se explican las bases teóricas y funcionamiento del test.

4.2.1. Procesos principales y componentes.

En la ilustración 29 se muestra una versión simplificada del circuito con los bloques principales para el funcionamiento. En este apartado se describen las características principales de estos componentes.

Para empezar se ha incluido en el diseño una librería de trabajo, *deftipos*, que guarda todos los valores de las constantes calculados en el capítulo de desarrollo 3.2.

Estos valores son *L, Q, Bl=K+Q, Smin, Smax* y las memorias POS y CONT que contienen la información necesaria para obtener el logaritmo en base 2 de 1 a 64640 (*Q+K).*

Contador de índice – INDICE.

Este contador se encarga de señalizar el bloque de datos de *L* bits que se está analizando y se incrementa en 1 por cada ciclo de reloj. La señal START habilita el funcionamiento del contador después de que se haya inicializado. Cuando el contador se incrementa de 1 a *Q+K,* se activa la señal GUARDAR que se encarga de habilitar la lectura y escritura de la memoria RAM que forma TABLA. Al pasar la etapa de inicialización (*i>Q*), se pone a nivel alto la señal ESTADO_K que indica el periodo durante el cual el SUM realiza el sumatorio de los logaritmos. Al finalizar la etapa de análisis se activa la señal FIN.

En este proceso intervienen distintas señales de entrada externas o de otros componentes, y se generan una serie de señales para controlar otros procesos posteriores:

CLK	Señal de entrada de reloj externa. Debe ser la misma que utilice el PRGN que se desea analizar y de una frecuencia máxima de 50Mhz.
RESET	Señal de entrada de inicialización de AUXI, GUARDAR, ESTADO_K y FIN.
CLEAR	Señal de inhabilitación procedente del circuito de seguridad.
START	Señal de entrada que inicia y habilita el contador.
AUXI	Señal que indica el valor de la cuenta y se incrementa en cada ciclo de reloj, su valor está comprendido entre 0 y 64641.
GUARDAR	Señal de habilitación para la memoria RAM TABLA, cuando *i* se incrementa de 1 a *Q+K*.
ESTADO_K	Señal que indica con su nivel alto la duración de la etapa de análisis, cuando *i* está comprendido entre *Q* y *Q+K.*
FIN	Señal que indica que el análisis estadístico ha finalizado si es igual a ‘1’.

El diagrama esquemático del circuito de muestra a continuación:

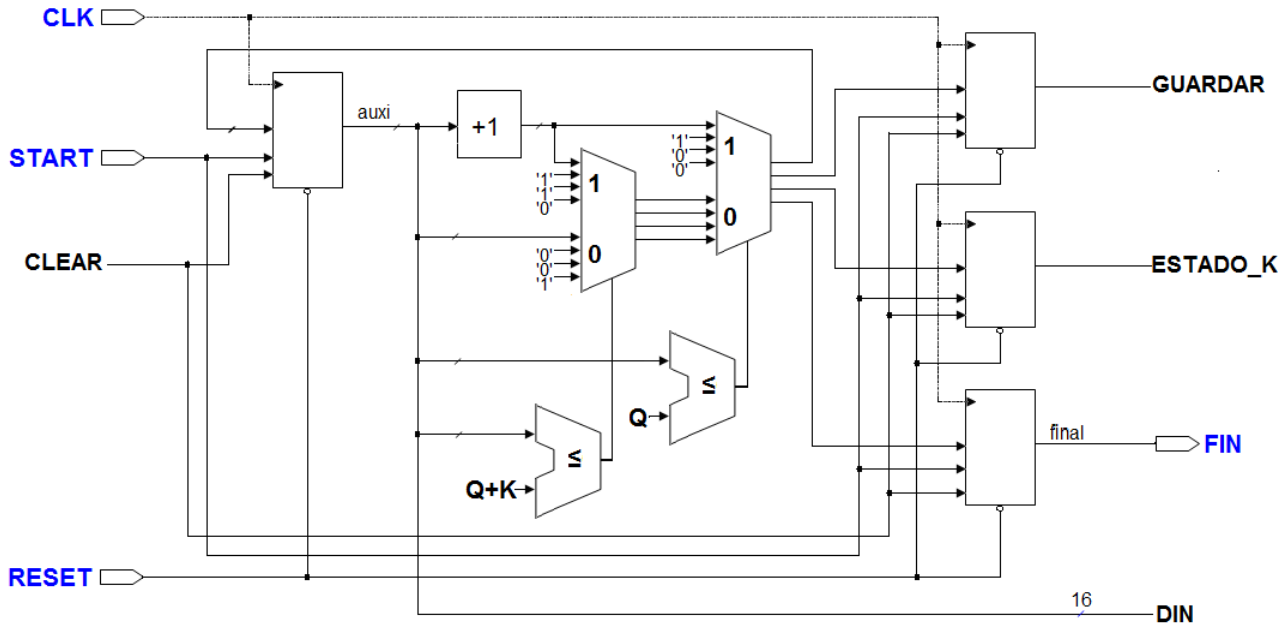


Ilustración 30. Diagrama esquemático del componente INDICE.

Para ver gráficamente el comportamiento de las señales se realizan las simulaciones de las distintas etapas:

- *I≤Q:* Se observa la inicialización de los registros y el aumento de la señal AUXI.

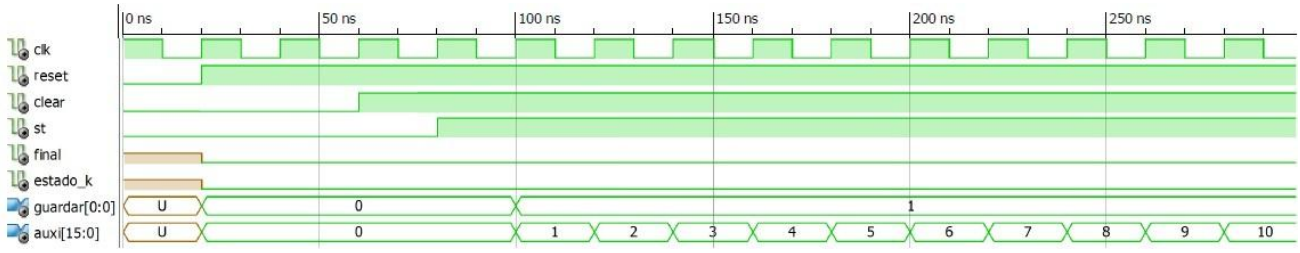


Ilustración 31. Simulación de INDICE de 0ns a 300ns.

- *I>Q* e *i≤Q+K:* Se muestra que a partir del bloque número *Q,* se activa la señal ESTADO_K que activa la etapa de análisis.

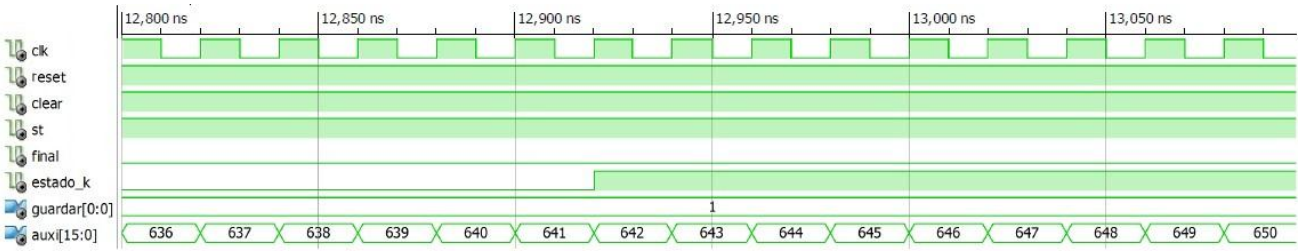


Ilustración 32. Simulación de INDICE de 12800 ns a 13100 ns.

- *I>Q+K:* Una vez que ya se han analizado todos los bloques se activa la señal FINAL y se desactiva la señal GUARDAR que evita que se sigan guardando datos en la memoria TABLA.

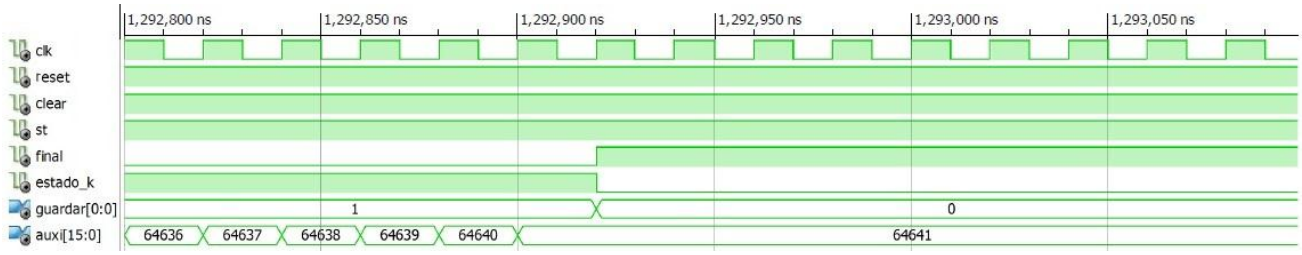


Ilustración 33. Simulación de INDICE de 1292800 ns a 1293100 ns.

Memoria RAM 64x16bits – TABLA.

Este componente guarda la posición donde se repite un bloque específico y por ello tiene las dimensiones de 64x16. Como los datos criptográficos que se desean analizar tienen un tamaño de 6 bits, existen 64 valores posibles para estos datos, que se han de ir repitiendo durante todo el análisis.

Por otro lado, como se llegan a probar hasta 64640 (*Q+K)* bloques de datos, el valor que se debe guardar ha de tener el tamaño suficiente para guardar esta cifra, es decir, 16 bits. Estos parámetros son específicos para las características del test y se detallan en el capítulo 3.2.

Para la optimización del componente TABLA se ha elegido crear una memoria RAM de 64 direcciones con 16 bits. Este componente se configura fácilmente utilizando otra de las herramientas que ofrece el paquete ISE de Xilinx, el CORE Generator. Con este programa se pueden crear distintos bloques funcionales cuyas características están predefinidas y optimizadas para aprovechar mejor los recursos.

Los puertos de entrada y salida de la memoria son:

CLKA	Señal de entrada de reloj. Es la misma que en el resto del circuito
WEA	Señal que habilita la escritura de la memoria. Se corresponde con la señal GUARDAR que se genera en el proceso INDICE.
ADDRA[5:0]	Señal que indica la dirección de memoria que se quiere leer o escribir. Es directamente la entrada de datos criptográficos S_N.
DINA[15:0]	Dato de entrada de la memoria cuando está escribiendo. Este dato se grabará en la posición de memoria designada por ADDRA en un flanco de subida de reloj. Este correspondiente a la señal AUXI, es decir, al número del bloque que se está analizando.
DOUTA[15:0]	Dato de salida correspondiente a la lectura de una determinada dirección señalada por ADDRA. Este valor sirve para saber en qué posición se repitió por última vez el valor S_N, que es el mismo que ADDRA. Para poder usar este dato se debe configurar la memoria en modo de lectura primero y así, no sobre escribir antes de leer el dato.

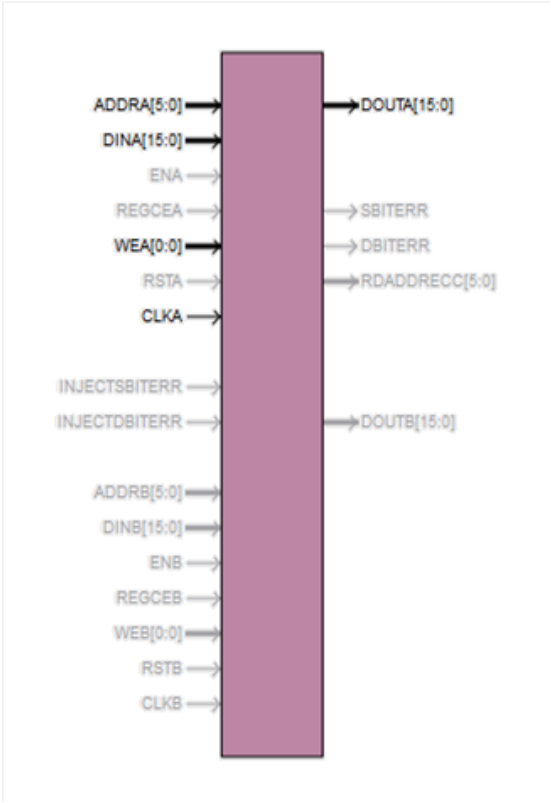


Ilustración 34. Bloque de memoria RAM de 64x16.

En la simulación siguiente se muestra el tramo inicial de configuración, donde se muestra como se escribe y se lee la memoria RAM:

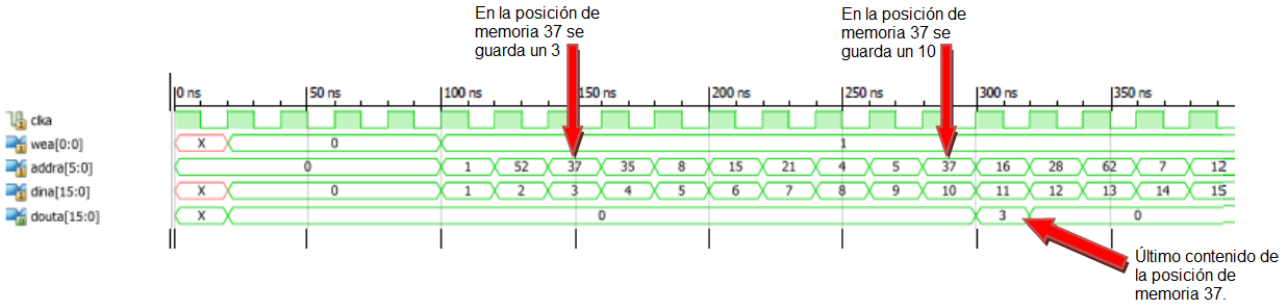


Ilustración 35. Simulación del componente RAM64_16.

Utilizando la entrada DINA y la salida DOUTA se puede calcular instantáneamente la diferencia entre la posición actual de un bloque determinado y la última vez que apareció. Por ejemplo, en la ilustración 35 se observa que el dato 37 (ADDRA) que aparece en la posición 10 (DINA), se repitió en la posición 3 (DOUTA en el siguiente ciclo de reloj). Por lo que la diferencia entre estos valores se puede hallar con una sentencia

concurrente en VHDL que reste DINA-DOUTA-1=C; para este ejemplo sería 11 - 3 – 1 = 7, respectivamente.

Búsqueda de logaritmos – LOG.

Una vez obtenido el valor de la diferencia entre repeticiones del mismo dato de la muestra aleatoria, C, se debe buscar el valor correspondiente de logaritmo de C en la memoria de logaritmos del 1 al 64640 (*Q*+*K*).

Uno de los criterios tomados durante el capitulo 3.2 fue recudir el valor de la memoria de logaritmos. Esta disminución se consiguió creando 2 memorias de 136 valores que se relacionan de la siguiente manera:

- POS, contiene el primer valor donde se empieza a repetir un determinado valor del logaritmo.
- CONT, resultado del logaritmo en base 2 de los valores comprendidos entre el valor correspondiente de POS y el siguiente. Es decir, $\log_2$[POS(index):POS(index+1)]=CONT(index), siendo *index* un valor comprendido entre el 1 y el 136.

El algoritmo de búsqueda consiste en encontrar a que subíndice pertenece el valor C del que hay que hallar el logaritmo. Por lo tanto, para un determinado C que está comprendido entre los valores de POS(index) a POS(index+1), el $\log_2$(C)=CONT(index).

Para la implementación hardware de este proceso se ha realizado un bucle que genera una red de multiplexores y compradores anidados para la obtención de la señal INDEX.

El bucle se describe en el diagrama de flujo de la ilustración 36:

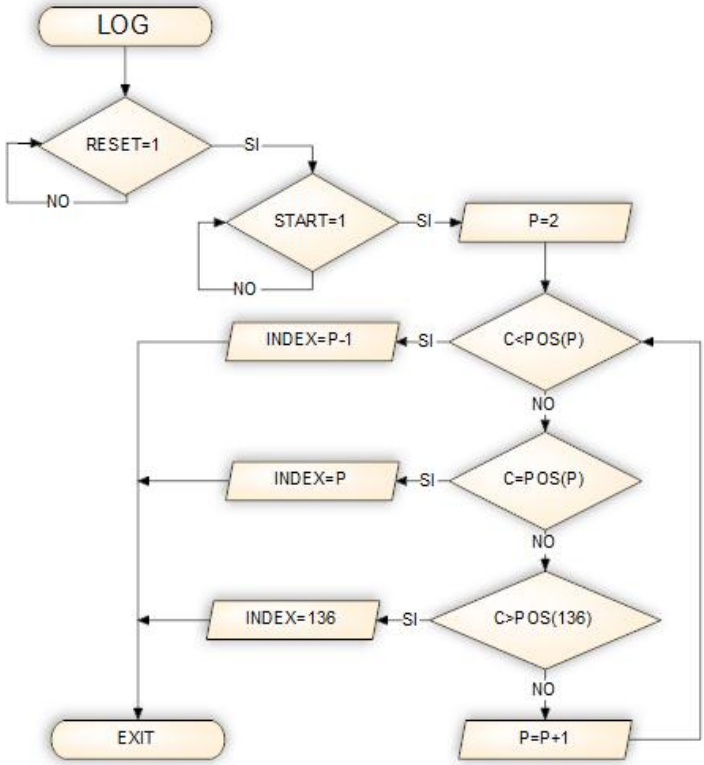


Ilustración 36. Diagrama de flujo del proceso LOG.

La simulación muestra el valor de C y el INDEX correspondiente que va un ciclo de reloj retrasado. Por ejemplo, el primer valor de C distinto de 1 es 50, y el valor de INDEX correspondiente es 32, esto quiere decir que el logaritmo en base 2 de 50 coincide con el valor numero 32 de la memoria CONT, $\log_2(50)=CONT(32)$.

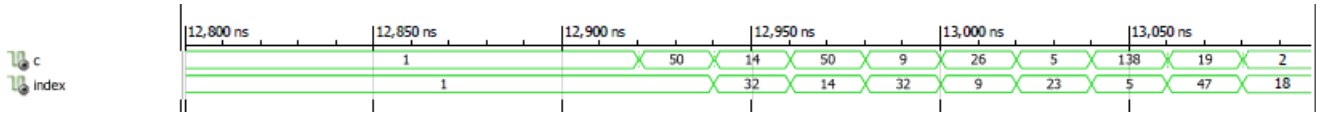


Ilustración 37. Simulación de LOG.

Sumatorio –SUM.

El paso final para completar el test estadístico es realizar el sumatorio de los logaritmos durante el periodo en el que la señal ESTADO_K del componente INDICE está a nivel alto.

Para realizar este proceso se utiliza un sumador síncrono cuya entrada es el valor de la memoria CONT correspondiente a la posición que indica INDEX., con una señal de habilitación ESTADO_K. La salida es la señal SUMA que cambia cada ciclo de reloj hasta que la señal FIN este a nivel alto. Una vez que se haya terminado el análisis de todos los bloques, la señal SUMA se compara con los valores de las constantes *Smin y Smax,* si dicho valor se encuentra dentro de estos límites la muestra se considera aleatoria y la

señal ALEATORIA_TEST se activa con un '1' lógico. El resultado del test también se puede leer por el puerto de salida VALUE_TEST.

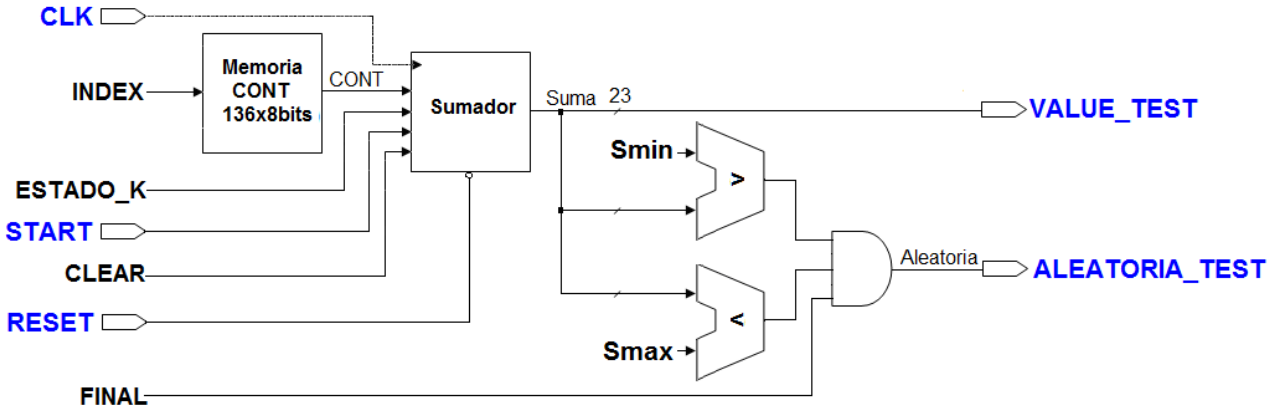


Ilustración 38. Circuito esquemático del proceso SUM.

Las simulaciones muestran el comportamiento de estas señales al inicio del periodo de análisis y al final:

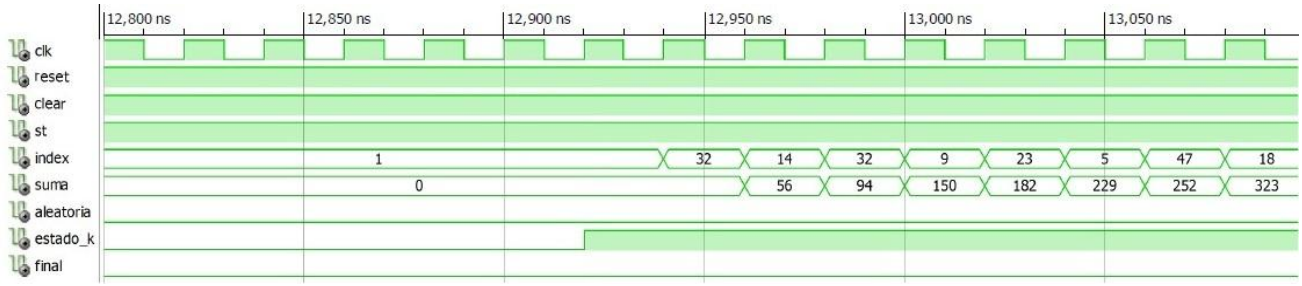


Ilustración 39. Simulación del proceso SUM de 12800ns a 131000ns.

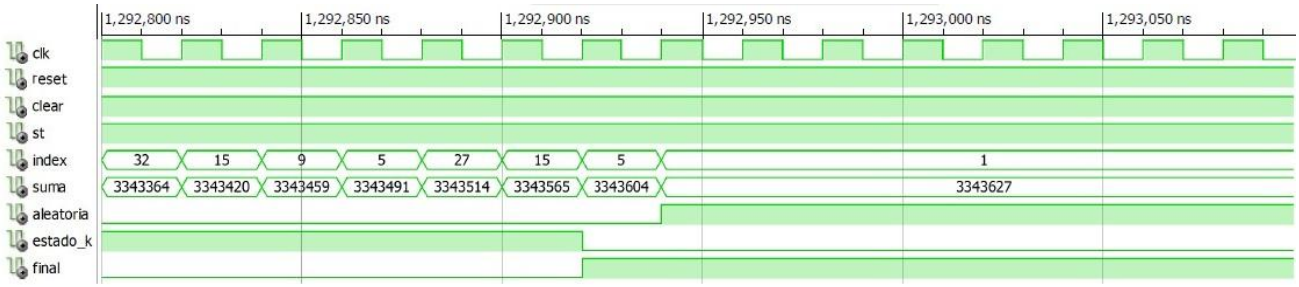


Ilustración 40. Simulación del proceso SUM de 1292800ns a 1293100ns.

En el capítulo 5 se muestran los resultados del circuito completo para distintas muestras aleatorias y no aleatorias.

4.2.2. Síntesis del circuito completo.

Una vez completado el diseño hardware del test estadístico se le añade el componente de seguridad desarrollado en el capítulo 4.1 y se sintetiza para comprobar los recursos que utiliza de la FPGA y las limitaciones de tiempo para el circuito completo. A continuación se muestra una copia de los apartados más importantes del Informe de Síntesis que se genera al realizar la implementación completa del circuito:

```
Device utilization summary:
---------------------------
Selected Device : 6slx9ftg256-3
Slice Logic Utilization:
 Number of Slice Registers:              80  out of  11440      0%
 Number of Slice LUTs:                  440  out of   5720      7%
    Number used as Logic:               437  out of   5720      7%
    Number used as Memory:                3  out of   1440      0%
       Number used as SRL:                3
Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:    460
   Number with an unused Flip Flop:     380  out of    460     82%
   Number with an unused LUT:            20  out of    460      4%
   Number of fully used LUT-FF pairs:    60  out of    460     13%
   Number of unique control sets:         9
IO Utilization:
 Number of IOs:                          43
 Number of bonded IOBs:                  43  out of    186     23%
Specific Feature Utilization:
 Number of Block RAM/FIFO:                1  out of     32      3%
    Number using Block RAM only:          1
 Number of BUFG/BUFGCTRLs:                2  out of     16     12%
```

```
Timing Summary:
---------------
Speed Grade: -3
   Minimum period: 10.525ns (Maximum Frequency: 95.009MHz)
   Minimum input arrival time before clock: 4.134ns
   Maximum output required time after clock: 3.984ns
   Maximum combinational path delay: 0.000ns
```

```
Found area constraint ratio of 100 (+ 5) on block test_9, actual
ratio is 9.
```

La FPGA Spartan 6 ofrece un soporte adecuado a las características del diseño y además es una opción que garantiza un bajo coste a baja potencia. La velocidad de análisis dependerá de la frecuencia que marque el generador de números aleatorios y está limitada a un máximo de 95Mhz. El análisis se desarrolla en aproximadamente 64.640

ciclos de reloj con un mínimo periodo obtenido de 10,525ns estiman un tiempo mínimo para el análisis completo de aproximadamente 0,7ms.

5. Simulación y pruebas.

En este capítulo se siguen todos los pasos necesarios para comprobar el funcionamiento del circuito en simulaciones y pruebas de laboratorio.

5.1. Procedimiento de pruebas y simulación.

Los siguientes apartados describen las instrucciones a seguir para realizar las pruebas del sistema. Cada sección está explicada para que se pueda simular el funcionamiento o probar en el laboratorio.

5.1.1. Obtención del valor DNA.

El primer paso es la obtención del DNA único para cada FPGA.

Si se quiere realizar una **simulación**, se debe elegir un valor aleatorio de DNA respetando las restricciones de este parámetro:

- El primer dígito en hexadecimal debe ser 0 y el segundo un valor del 0 al 7, para que la trama de bits sea correcta.
- Este valor tiene una longitud de 15 cifras expresadas en hexadecimal, por ejemplo, Dna_sim_value: X"0123456789ABCDE".

Se completa directamente el campo genérico DNA_SIM_VALUE del componente DNA_PORT en el fichero *dna.vhd* perteneciente al circuito de seguridad. Este valor será el valor identificativo de la FPGA simulada.

En el caso de trabajar con una FPGA real en las **pruebas de laboratorio**, se obtiene el DNA de la FPGA conectando la tarjeta de pruebas al PC por el puerto JTAG. Desde la ventana de comandos de Windows se ejecuta el fichero *rddna.cmd* que extrae el valor identificativo DNA y lo muestra por pantalla (ver el Anexo I). Este valor se muestra en binario y tiene una longitud de 57 bits.

5.1.2. Cálculo del valor de memoria ROM.

Con el fin de añadir un nivel de dificultad al circuito de seguridad, no se realiza la autocomprobación con el valor auténtico de DNA sino que se modifica con un algoritmo de encriptación sencillo.

Independientemente de si el valor de DNA es simulado o real, el siguiente paso es calcular el valor de DNA encriptado que se guardará en la memoria ROM. Para ello se hace uso de una plantilla creada en Excel en la que solo es necesario completar el campo DNA.

Plantilla 1. Obtención del valor de memoria ROM.

Esta plantilla en Excel está diseñada para la obtención del valor que se debe guardar en la memoria ROM para el funcionamiento correcto del circuito de seguridad. Este valor es el resultado de modificar mediante un algoritmo de seguridad el valor único de la FPGA, es decir, el DNA.

Con el uso de esta plantilla de obtienen los campos necesarios para la simulación y además, es un paso previo para una segunda plantilla que obtiene los datos binarios que habría que cambiar en el bitstream para modificar la memoria ROM sin necesidad de volver a implementar el circuito.

Las instrucciones de uso se enumeran a continuación:

1. Se debe extraer el valor del DNA de la FPGA o elegir un valor en caso de simulación.
2. Los 8 primeros bits (d63 a d57) de la columna DNA no se modifican en la plantilla y no pertenecen al valor del DNA.
3. Cambiar en la plantilla la zona sombreada de azul con el valor binario del DNA. En el caso del valor de simulación que tiene 15 dígitos hexadecimales, el primer dígito siempre es 0 y se desprecia al rellenar esta plantilla. En su caso siempre se rellena el valor de d56 con un ‘1’. En la casilla a la derecha aparecerá el valor en hexadecimal correspondiente para la comprobación y poder evitar fallos.

PLANTILLA 1 (modificar el campo DNA)

	sScr	DNA	Dna hex	ROM	Rom hex		sScr	DNA	Dna hex	ROM	Rom hex
d63	**0**	1		1		d31	0	0		0	
d62	**1**	1	F	0	8	d30	0	1	5	1	4
d61	**1**	1		0		d29	0	0		0	
d60	**1**	1		0		d28	1	1		0	
d59	**1**	0		1		d27	1	0		1	
d58	**1**	0	1	1	D	d26	0	0	0	0	9
d57	**0**	0		0		d25	0	0		0	
d56	**0**	1		1		d24	1	0		1	
d55	**0**	0		0		d23	0	1		1	
d54	**0**	0	1	0	3	d22	1	0	8	1	D
d53	**1**	0		1		d21	0	0		0	
d52	**0**	1		1		d20	1	0		1	
d51	**0**	0		0		d19	1	1		0	
d50	**0**	1	6	1	5	d18	1	0	B	1	4
d49	**1**	1		0		d17	1	1		0	
d48	**1**	0		1		d16	1	1		0	
d47	**0**	0		0		d15	0	1		1	
d46	**0**	1	7	1	5	d14	0	0	9	0	9
d45	**1**	1		0		d13	0	0		0	
d44	**0**	1		1		d12	0	1		1	
d43	**1**	1		0		d11	1	0		1	
d42	0	1	E	1	5	d10	0	0	0	0	8
d41	1	1		0		d9	0	0		0	
d40	1	0		1		d8	0	0		0	
d39	1	0		1		d7	1	1		0	
d38	1	1	7	0	9	d6	1	1	F	0	3
d37	1	1		0		d5	0	1		1	
d36	0	1		1		d4	0	1		1	
d35	0	0		0		d3	1	1		0	
d34	0	0	2	0	3	d2	0	1	C	1	6
d33	0	1		1		d1	1	0		1	
d32	1	0		1		d0	0	0		0	

Tabla 14. Plantilla 1.

DNA_sim_value	**0167E72508B90FC**
ROM	
INIT (SRLC32E_1)->rom1	**8D355593**
INIT (SRLC32E_0)->rom2	**49D49836**

Tabla 15. Tabla de resultados de la Plantilla 1.

En la tabla de resultados están los valores que hay que introducir en la memoria ROM del circuito de seguridad, únicos para cada dispositivo FPGA y que se pueden usar para modificar directamente el diseño en VHDL para la simulación. En el siguiente apartado se describe como se modifica el circuito para cambiar el valor de la memoria ROM.

5.1.3. Modificación del circuito con el valor ROM calculado.

Una vez obtenido el valor de la memoria ROM se debe modificar este parámetro en el circuito para que la comprobación que posteriormente realiza de forma automática el circuito sea correcta.

Si se está realizando una **simulación**, se debe completar el campo INIT del componente SRLC32E_1 y SRLC32E_0 descritos dentro del fichero *rom.vhd.* En la tabla de resultados que genera la Plantilla 1 se muestra claramente el contenido de cada uno y lo único que hay que hacer es copiar y pegar. Al completar este campo ya estaría realizado el proceso de configuración y se podría simular el banco de pruebas *tb.vhd*.

Plantilla 2 de codificación del valor ROM para modificar el Bitstream base.

En el caso de estar efectuando **pruebas de laboratorio**, el proceso requiere una información más detallada:

1. Previamente, se ha realizado la implementación completa del diseño con un valor de memoria ROM igual a “0000000000000000”. Se genera el fichero de programación de la FPGA, es decir, el bitstream base o plantilla, que se denomina con el nombre *test.bit.*
2. Se utiliza la Plantilla 2 que ya muestra en el campo ROM el valor obtenido por la Plantilla 1. Se cambian en la plantilla la zona sombreada de azul con los datos en hexadecimal que se indican en la zona superior de la plantilla y corresponden con el valor ROM calculado.
3. Abrir desde la ventana de comandos de Windows el programa VBINDIFF y el fichero test.bit. *(>>vbindiff.exe test.bit*).
4. Modificar los datos resaltados de la tabla de resultados de la Plantilla 2, tabla 17.
   - Para buscar la fila correspondiente pulsar la tecla “G”, se abrirá una pequeña ventana para escribir la fila donde se desea modificar un dato.
   - Para editar el fichero pulsar la tecla “E”, y utilizar las flechas del teclado para moverse entre las columnas. Una vez hecha la modificación pulsar “ESC”, y “Y” al guardar los cambios.

Ilustración 41. Interfaz de comandos del programa VBINDIFF.

5. La modificación del fichero se ha realizado. El único paso que queda es descargar el bitstream *test.bit* en la FPGA y probarlo en el laboratorio.

PLANTILLA 2 (modificar d0 a d15)

ROM: X " 8D35559349D49836 "

Fila	Columna		d0	d1	d2	d3	d4	d5	d6	d7	d8	d9	d10	d11	d12	d13	d14	d15
			8	D	3	5	5	5	9	3	4	9	D	4	9	8	3	6
0001 E130	10	C3															3	C
0001 E130	11	3C													C	3		
0001 E130	12	0C											C	0				
0001 E130	13	30									0	3						
0001 E140	2	CC							C	C								
0001 E140	3	00					0	0										
0001 E140	4	03			3	0												
0001 E140	5	3C	C	3														
0001 E1B0	12	33															3	3
0001 E1B0	13	03													3	0		
0001 E1B0	14	3F											F	3				
0001 E1B0	15	CC									C	C						
0001 E1C0	4	C3							3	C								
0001 E1C0	5	FF					F	F										
0001 E1C0	6	F3			3	F												
0001 E1C0	7	F0	0	F														

Dato Tipo A	0	0	3	3	0	0	3	3	C	C	F	F	C	C	F	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	3	0	3	C	F	C	F	0	3	0	3	C	F	C	F
Dato Tipo B	0	0	C	C	0	0	C	C	3	3	F	F	3	3	F	F
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	0	C	0	C	3	F	3	F	0	C	0	C	3	F	3	F

Tabla 16. Plantilla 2.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0001 E130	XX	XX	XX	XX	XX	XX	XX	XX	XX	C3	3C	0C	30	XX	XX	XX
0001 E140	XX	CC	00	03	3C	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX
0001 E1B0	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	XX	33	03	3F	CC	XX
0001 E1C0	XX	XX	XX	C3	FF	F3	F0	XX	XX	XX	XX	XX	XX	XX	XX	XX

Tabla 17. Tabla de resultados de la Plantilla 2.

5.1.4. Obtención de muestras aleatorias y no aleatorias.

Este paso solo tiene importancia en las simulaciones ya que es la forma en que se generan muestras del tamaño específico necesario en el test para utilizarlas en el banco de pruebas (ver Anexo I).

Para ello se ha utilizado MatLab y se han generado dos tipos de muestras de un tamaño de 400000 bits:

- Muestras aleatorias: se generan con la función *rand(1,400000).* El resultado es una muestra de 400000 valores decimales del 0 al 1. Para conseguir la muestra binaria se redondean los datos a las unidades.
- Muestra no aleatoria: se crea un vector de 400000 ceros o unos y se introducen algunas diferencias repetitivas cada cierto número de bits.

Estas muestras se introducen en el banco de pruebas *tb.vhd* y se relacionan con la entrada de datos S_N.

5.2. Resultados.

A continuación se irán mostrando las simulaciones del circuito de seguridad para comprobar los resultados de las Plantillas1 y 2. Por otra parte se prueba el circuito completo con el test estadístico necesarias para la comprobación de los resultados obtenidos con MatLab.

5.2.1. Simulación y pruebas del circuito de seguridad.

Como se ha descrito anteriormente la señal RST del componente de seguridad es la encargada de inhabilitar el dispositivo FPGA, en el caso de que la comparación del valor forzado en la Memoria ROM no coincida con el del dispositivo encriptado. Si el dispositivo no ha sido manipulado esta señal se activará a nivel alto.

Para comprobar el correcto funcionamiento del circuito de seguridad se realizaran tres simulaciones, una con el valor correcto y las otras dos con valores diferentes de la variable genérica DNA_SIM_VALUE que está predeterminada en el componente de Xilinx. Lo cuál nos permite simular que tenemos 3 FPGAs distintas.

Se siguen los pasos detallados en el capítulo 5.1 para cada FPGA simulada. El último paso para realizar la simulación es, una vez obtenidos todos los datos, guardar y simular. Para ver los resultados con mayor claridad utilizar al fichero Sim1.wcfg.

El valor seleccionado en la simulación es:

Dna_sim_value: X"0167E72508B90FC"

Para hallar el valor que debemos guardar en la memoria ROM se utiliza la Plantilla 1 siguiendo los pasos a continuación:

a.- Se desprecia el primer dígito hexadecimal, que debe ser 0.

b.- Se introduce el valor binario de cada dígito hexadecimal en la zona sombreada de azul claro.

c.- Se comprueba que el valor de DNA introducido es correcto con el valor en Hexadecimal de la columna adyacente a la derecha.

d.- El valor necesario en la memoria ROM, se copia directamente del resultado resaltado en amarillo a los registros de desplazamiento SRLC32E_0 y SRLC32E_1.

Valores correctos obtenidos a partir de la Plantilla 1.

Dna_sim_value: X"0167E72508B90FC"

Rom: X"8D35559349D49836"

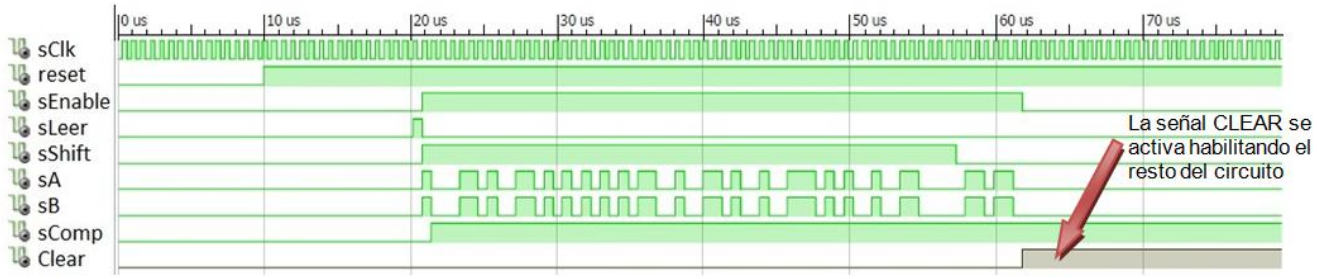


Ilustración 42. Simulación de comprobación correcta.

Cambiamos el primer dígito del Dna_sim_value, simulando que es otro dispositivo FPGA (copia en otra FPGA).

Dna_sim_value: X"0**5**67E72508B90FC"

Rom: X"8D35559349D49836"

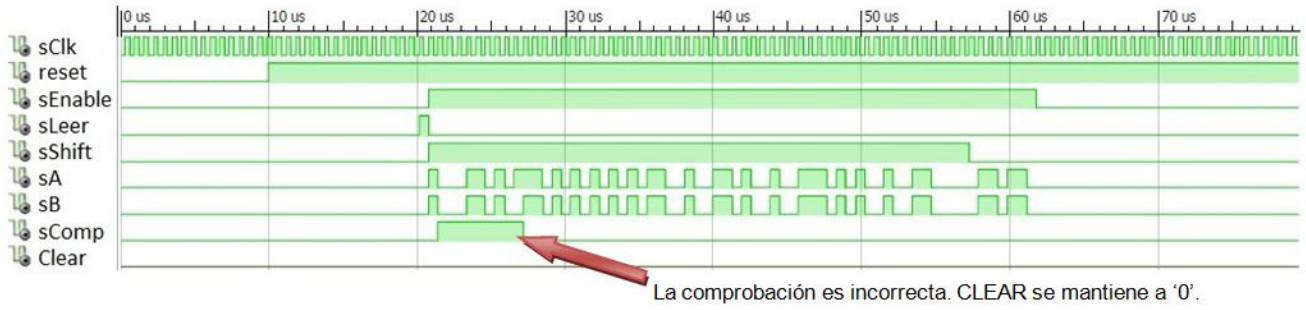


Ilustración 43. Simulación de copia 1.

Cambiamos el último dígito del Dna_sim_value, simulando que es otro dispositivo FPGA, distinto a los dos anteriores.

Dna_sim_value: X"0167E72508B90F**0**"

Rom: X"8D35559349D49836"

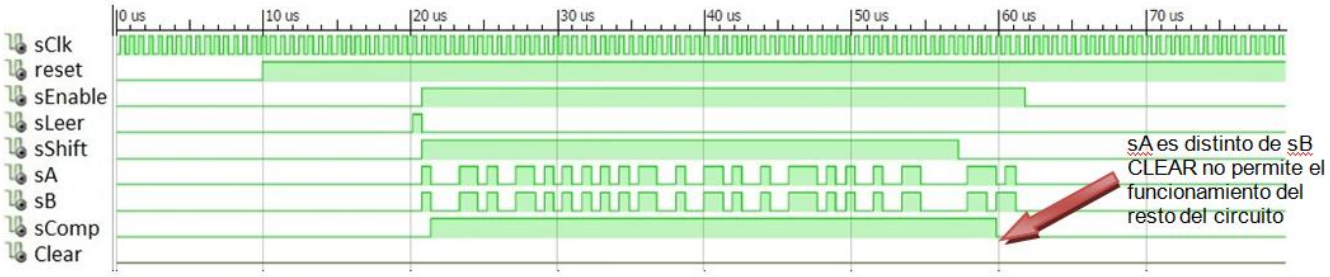


Ilustración 44. Simulación de copia 2.

Para probar el circuito de seguridad en el laboratorio existe una limitación: sólo se pueden probar los resultados sobre una FPGA. Siguiendo las instrucciones del capítulo 5.1 específicas para **pruebas en el laboratorio**, se obtienen los siguientes resultados para la FPGA Spartan 6 específica:

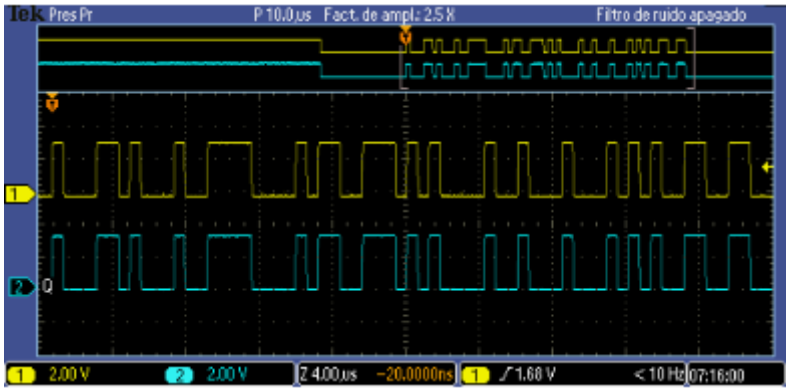


Ilustración 45. Pruebas de laboratorio (sA=sB).

La salida se analiza de forma visual mediante el LED de la tarjeta y los resultados son correctos señalando que la señal de salida CLEAR funciona correctamente.

Ya que no se puede probar el ejemplo de copia sobre otra tarjeta, se opta por realizar una modificación del fichero de configuración plantilla con un valor distinto al obtenido con el procedimiento de pruebas. La comparación es correcta hasta casi el final de la comparación. En esta prueba el LED correspondiente a la señal CLEAR no se enciende. Los resultados se muestran en la ilustración 46.

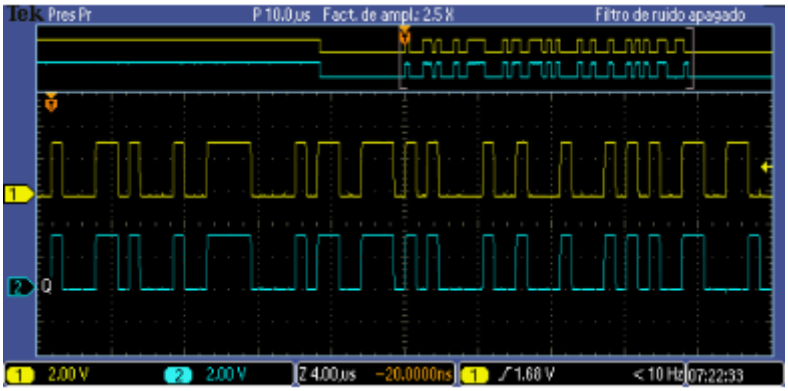


Ilustración 46. Pruebas de laboratorio (sA≠sB).

5.2.2. Simulación del circuito completo con muestras aleatorias y no aleatorias.

Una vez comprobado el funcionamiento del circuito de seguridad, se simula el comportamiento del diseño completo ante distintas muestras criptográficas. El tiempo necesario para realizar el análisis completo de la muestra es de aproximadamente 1.3 ms, como se observó en las simulaciones realizadas en el capítulo 4.2.

La primera simulación muestra el caso de funcionamiento si el circuito se manipula y se copia sobre otra FPGA distinta de la original. Al no activarse la señal CLEAR el circuito no actúa independientemente del valor de las entradas. Las señales de salida FIN, ALEATORIA y VALUE_TEST no se modifican.

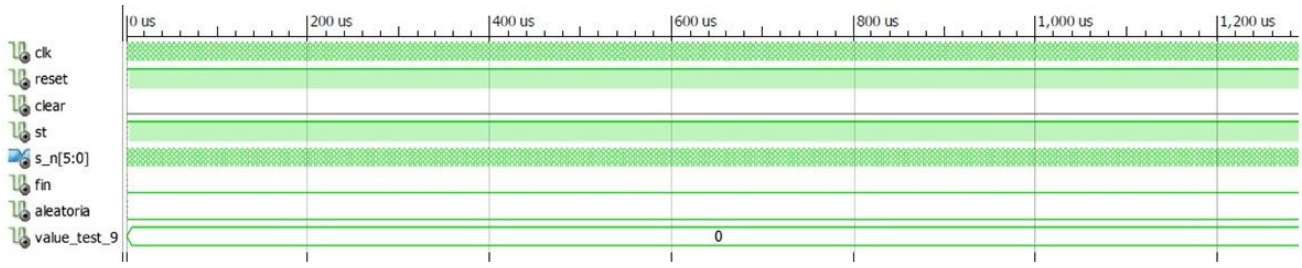


Ilustración 47. Simulación del circuito completo en una FPGA distinta a la original.

Las siguientes simulaciones muestran los puertos de entrada y salida del circuito completo, también se han añadido las señales CLEAR y RESET para facilitar la comprensión de los resultados.

En los apartados correspondientes del capítulo 4 de implementación se ha simulado y descrito el funcionamiento de cada señal. El objetivo de estas simulaciones es comprobar la configuración inicial de la señal CLEAR y el resultado final que muestra la señal VALUE_TEST_9.

TB 1 (muestra aleatoria)

Ilustración 48. Simulación del circuito completo con la muestra aleatoria 1.

TB 2 (muestra aleatoria)

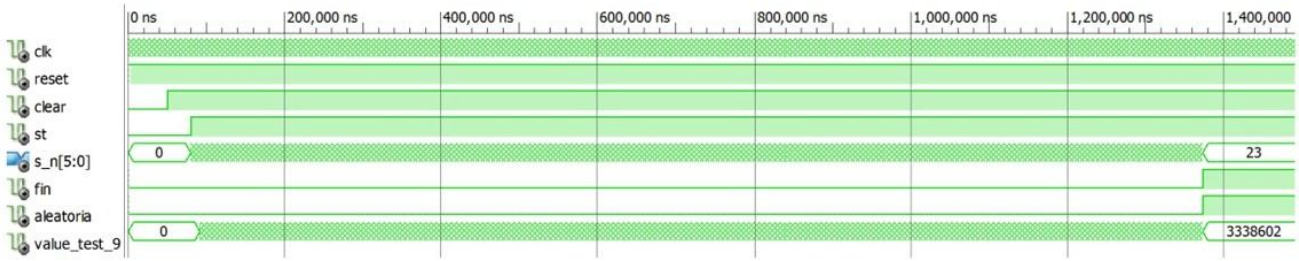


Ilustración 49.Simulación del circuito completo con la muestra aleatoria 2

TB 3 (muestra no aleatoria)

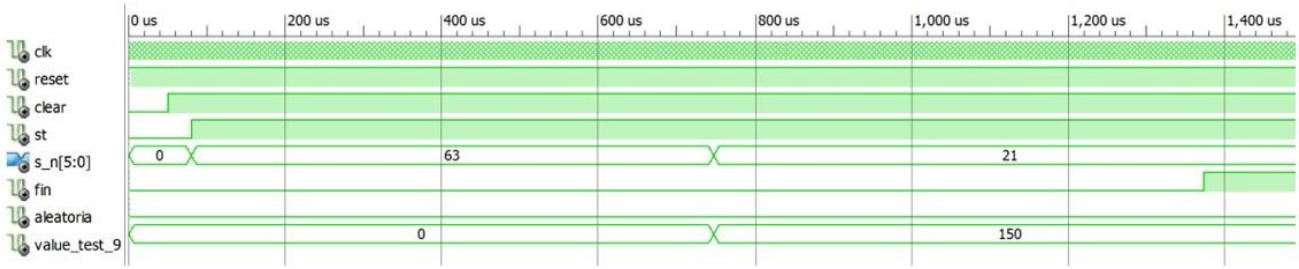


Ilustración 50. Simulación del circuito completo con la muestra no aleatoria 3.

TB 4 (muestra no aleatoria)

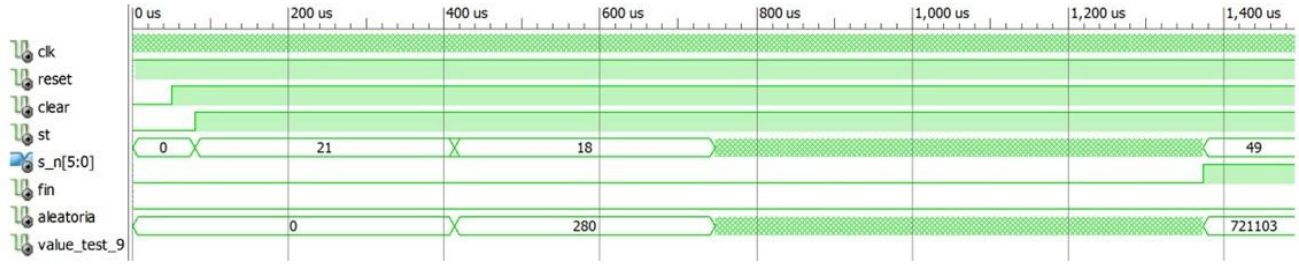


Ilustración 51. Simulación del circuito completo con la muestra no aleatoria 4.

Todas las simulaciones indican un funcionamiento correcto y previsible de las salidas. Con el fin de verificar cuantitativamente los resultados obtenidos se procede al cálculo total de los errores absolutos y relativos para las 4 simulaciones. El valor *Sum,* que se muestra en la primera fila de la Tabla 18 pertenece al valor calculado con la herramienta matemática MatLab para valores de logaritmo con 16 decimales.

	TB 1	**TB 2**	**TB 3**	**TB 4**
***Sum* (16 decimales)**	3342671	3337643	150	721365
VALUE_TEST_9	3343627	3338602	150	721103
Error absoluto	956	959	0	262
Error relativo	$2{,}86 \times 10^{-4}$	$2{,}87 \times 10^{-4}$	0	$3{,}63 \times 10^{-4}$
Error relativo (%)	0,03	0,03	0	0,04

Tabla 18. Error relativo y absoluto del circuito de test estadístico.

Con un error relativo medio de del 0.025%, se puede asegurar la fiabilidad del test en 99.97%.

6. Conclusión.

El objetivo principal de diseñar un sistema de verificación de generadores de números aleatorios o pseudo-aleatorios empleando un diseño electrónico. Ha resultado una método eficaz y con una fiabilidad del 99.97% como resaltan los resultados finales comparados con las pruebas realizadas con MatLab. El dispositivo FPGA utilizado ha ofrecido un buen soporte para la realización completa del circuito y gracias a su flexibilidad se ha podido diseñar un circuito de bajo coste a medida.

El tamaño de las muestras que se han analizado excede los 387.000 bits, obteniendo un resultado de tiempo de análisis aproximado de 0,7 ms en la implementación hardware. En el análisis hecho mediante el software del paquete de tests estadísticos del NIST [5] se estima un tiempo que puede variar de 1,4 a 14,4 segundos para muestras comprendidas entre 100.000 bits y 1.000.000 de bits. La velocidad computacional de estos dispositivos ha permitido que el tiempo necesario para el análisis estadístico sea inferior al obtenido por los diseños realizados en software, obteniendo una ventaja de tiempo de 4 órdenes de magnitud, tal y como se ha probado en el artículo [13].

Por otro lado, el circuito de seguridad en el arranque genera correctamente una señal que inhabilita el resto del circuito implementado en la FPGA y se ha logrado elaborar un procedimiento de pruebas explícito para reconfigurar el fichero de configuración de la FPGA, e individualizarlo para cada valor identificativo DNA. El sistema elegido de modificación del bitstream, directamente sobre el fichero binario, ofrece una ventaja frente al tiempo que se emplea en regenerar un nuevo fichero de configuración mediante entornos de desarrollo de las FPGAs. Las ventajas de esta aplicación de seguridad se basan principalmente en su sencillez, capacidad de acoplamiento para proteger otros diseños, su funcionamiento automático e independencia.

El proceso de diseño electrónico por parte de la Industria se separa del proceso de fabricación de las distintas tarjetas, es decir, cada tipo de industria desarrolla tarjetas que pueden integrar dispositivos FPGA, pero normalmente son otras compañías las que se encargan de la cadena de fabricación. Estas compañías de fabricación necesitan la información suficiente para crear el producto diseñado y en algunas ocasiones se puede producir la manipulación o copia de la propiedad intelectual en esta etapa de la producción. Al no ser necesario un entorno de diseño integrado para la reconfiguración de la FPGA, se puede suministrar, por parte de la compañía que desarrolla el diseño, únicamente el fichero de configuración y el procedimiento de reconfiguración. De esta forma no se difunde el diseño (ficheros VHDL, netlist, esquemáticos) de manera tan directa y evidente.

Una de las ampliaciones futuras que se pueden desarrollar para mejorar el proyecto es la realización de una aplicación que al conectar el PC con la FPGA durante el proceso

de configuración, lea automáticamente el valor DNA, lo encripte, reconfigure el bitstream plantilla y descargue el fichero de configuración modificado en la FPGA. De esta forma se automatizaría el procedimiento para pruebas que en el presente proyecto se plantea de forma manual.

Finalmente se estima que durante la evolución del proyecto se ha cumplido con los objetivos planteados inicialmente y se han resuelto satisfactoriamente los obstáculos presentados en el desarrollo.

7. Referencias.

[1] Xilinx. Design Security for High Volume Applications.

[2] NIST.AES. http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

[3] Xilinx. Partial Reconfiguration User Guide (UG702.pdf).

[4] Juan Quero Llor. Aceleración por Hardware de Algoritmos Basados en Soft-Computing Mediante Dispositivos Programables y Reconfiguración Dinámica.

[5] NIST. A statistical Test suite for Random and Pseudorandom Number generator for Cryptographic Applications, NIST special Publications. National Institute for Standards and Technology.

[6] G. Marsaglia. The Marsaglia Random Number CDROM, including the DIEHARD Battery of test of Randomness, Department of statistics, Florida State University. http://stat.fsu.edu/pub/diehard.html

[7] L'Ecuyery Simard. Paquete estadistico de tests TESTU01 (2007). http://www.iro.umontreal.ca/~simardr/testu01/tu01.html.

[8] Xilinx. Configuración de FPGA Spartan 6 (UG380.pdf, pag. 63).

[9] Xilinx. Configuración de FPGA Spartan 6 (UG380.pdf, pag. 104).

[10] Xilinx. User guide of PlanAhead.

[11] NIST. A statistical Test suite for Random and Pseudorandom Number generator for Cryptographic Applications, NIST special Publications. National Institute for Standards and Technology (Pag. 42).

[12] Xilinx. Sparatan 6 Datasheet (DS162.pdf).

[13] Anna Vaskova. Robust Cryptographic Ciphers with On-line Statistical Properties Validation.

8. Anexos.

8.1. Anexo I. Ficheros de desarrollo.

Rddna.cmd	setmode -bscan setcable -p auto identify readdna -p 1>temp.txt quit

Logaritmo.m	for i=1:1:64640,
	  x(i)=log2(i);
	end;
	w=roundn(10*x,0);
	save log.dat w -ascii
	
	n=1;
	pos(1)=1;
	cont(1)=w(1);
	for i=1:1:64639,
	  if w(i)~=w(i+1),
	  pos(n+1)=i+1;
	  cont(n+1)=w(i+1);
	  n=n+1;
	  else,
	  n=n;
	  end;
	end;
	pos;
	cont;
	save pos.dat pos -ascii
	save cont.dat cont -ascii

Limites.m	format long;
	L=6;
	K=64000;
	Q=640;
	eVL=5.21771;
	V=2.954;
	c=0.7+0.8/L+(4+32/L)*(K^(-3/L))/15;
	sigma=c*sqrt(V/K);
	A=2.3267*sqrt(2)*sigma;
	fmin=-A+eVL;
	fmax=A+eVL;
	smin=fmin*64000;
	smax=fmax*64000;
	Smin=roundn(10*smin,0)

	Smax=roundn(10*smax,0) save limites.dat Smin Smax -ascii

Muestra.m	x=rand(1,400000);
	s_n=roundn(x,0);
	
	save muestra.dat s_n -ascii

Muestra_na.m	%na_1
	x=ones(1,400000);
	for i=1:2:200001,
	  x(i)=0;
	end;
	
	%na_2
	% x=zeros(1,400000);
	% for i=1:5:200000,
	% x(i)=1;
	% x(i+1)=1;
	% end;
	% for i=200001:3:300003,
	% x(i)=1;
	%
	% end;
	% for i=300004:2:400000,
	% x(i)=1;
	% end;
	
	s_n=roundn(x,0);
	save muestra.dat s_n -ascii

Test.m	logaritmo
	limites
	muestra
	
	conv_q_k=zeros(64640,6);
	
	for i=1:1:64640,
	  conv_q_k(i,1:6)=s_n((400000-6*i+1):(400000-6*i+6));
	  b=dec2bin(conv_q_k(i,1:6));
	  ds = horzcat(b(1), b(2), b(3), b(4), b(5), b(6));
	  ti(i)=bin2dec(ds);
	
	end;
	
	tabla=zeros(64);
	
	suma=0;
	for i=1:1:64640,
	  if i<=Q,
	  tabla((ti(i)+1))=i;
	  else
	  a=i;
	  b=tabla((ti(i)+1));
	  c=a-b;
	  suma=suma+w(c);
	  tabla((ti(i)+1))=i;
	  end;
	end;
	
	if (suma>Smin) & (suma<Smax),

	aleatorio=1;
	else
	aleatorio=0;
	end;
	
	aleatorio
	suma
	
	save resultado.dat suma aleatorio -ascii

Errores.m	format long;
	for i=1:1:64640,
	x(i)=log2(i);
	w(i)=roundn(x(i),-1);
	Ea(i)=w(i)-x(i);
	Er(i)=Ea(i)/x(i);
	end;
	
	Eamax=max(Ea)
	Ermax=max(Er)

8.2. Anexo II. Ficheros de implementación hardware VHDL.

Test_9.vhd	library ieee;
	use ieee.std_logic_1164.all;
	use ieee.std_logic_arith.all;
	use ieee.std_logic_unsigned.all;
	Library XilinxCoreLib;
	use work.deftipos.all;
	
	entity test_9_2 is
	port (clk : in std_logic;
	reset : in std_logic;
	start : in std_logic;
	aleatoria_test9: out std_logic;--secuencia aleatoria
	value_test_9: out integer;
	fin : out std_logic; --fin del análisis
	s_n : in std_logic_vector (5 downto 0));
	
	end test_9_2;
	architecture a of test_9_2 is
	
	component ram64_16
	port (
	clka: IN std_logic;
	wea: IN std_logic_vector(0 downto 0);
	addra: IN std_logic_VECTOR(5 downto 0);
	dina: IN std_logic_VECTOR(15 downto 0);
	douta: OUT std_logic_VECTOR(15 downto 0));
	end component;
	
	component seguridad
	port (

```
            Rst: in std_logic;
          Clr : out  STD_LOGIC;
          Vida: out std_logic);
       end component;


  signal c    : integer range 0 to 64642;
  signal index : integer range 1 to 136;
  signal suma           : integer range 0 to 33554432; --
sumatorio de logaritmos
  signal aleatoria,estado_K,final: std_logic;
  signal guardar : std_logic_vector (0 downto 0);
  signal auxi           : std_logic_vector(15 downto 0);
      signal din,dout         : std_logic_vector(15 downto 0);

signal Clear: std_logic;
signal sVida: std_logic;

begin

TABLA : ram64_16
            port map (
                  clka => clk,
                  wea => guardar,
                  addra => s_n,
                  dina => din,
                  douta => dout);

SEGUR: seguridad
port map (Rst=>Reset,
                  Clr => Clear,
                  Vida => sVida);

INDICE: process(clk, reset)
begin
       if Reset = '0' then
            auxi<=(others=> '0');
            estado_K<='0';
            guardar<="0";
            final<='0';

       elsif clk'event and clk = '1' then
            if Clear='1' and Start='1'  then
                  if auxi<=Q then
                        auxi<=auxi+1;
--incremento de señal para el indice de las tablas
                        estado_K<='0';
                        guardar<="1";
                        final<='0';
                  elsif auxi<=Bl then
                        auxi<=auxi+1;
                        estado_K<='1';
                        guardar<="1";
                        final<='1';
                  else
                        auxi<=auxi;
                        final<='1';
                        estado_K<='0';
                        guardar<="0";
```

```
                  end if;
            end if;
   end if;
end process;

LOG: process(clk, reset)
begin
      if Reset = '0' then
            index<=1;
      elsif clk'event and clk = '1' then
            if Clear='1' and Start='1'  then
                  if estado_K='1' then
                        for p in 2 to 136 loop
--búsqueda de valores en la memoria de logaritmos
                              if c<LOG_POS(p) then
--índice correspondiente a c
                                    index<=p-1;
                                    exit;
                              elsif c=LOG_POS(p) then
                                    index<=p;
                                    exit;
                              elsif c>LOG_POS(136) then
                                    index<=136;
                                    exit;
                              end if;
                        end loop;
                  else
                        index<=1;
                  end if;
            end if;
      end if;
end process;

SUM: process(clk, reset)
begin
      if Reset = '0' then
            suma<=0;
            aleatoria<='0';
      elsif clk'event and clk = '1' then
            if Clear='1' and Start='1'  then
              suma<=suma+LOG_CONT(index);
--suma de logaritmos para cada indice calculado
              if final='1' and (suma>Smin and suma<Smax) then
                  aleatoria<='1';
              end if;
            end if;
   end if;
end process;

din<=auxi;
c<=conv_integer(din-dout-'1') when estado_K='1' else 1;

aleatoria_test9 <= aleatoria;
value_test_9<=suma;
fin<=final;

end a;
```

Seguridad.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use ieee.std_logic_unsigned.all;
Library UNISIM;
use UNISIM.VComponents.all;

entity seguridad is
    Port (  Rst: in std_logic;
            Clr : out  STD_LOGIC;
            Vida: out std_logic
              );
end seguridad;

architecture Behavioral of seguridad is

COMPONENT CLK
    Port ( CLOCK : out  STD_LOGIC;
            ES: out std_logic);

END COMPONENT;

component compare
    Port ( A : in  STD_LOGIC;
           B : in  STD_LOGIC;
           enable : in  STD_LOGIC;
            clk: in std_logic;
            Reset: in std_logic;
            Clear : out  STD_LOGIC);
end component;

component control
    Port ( Clk : in  STD_LOGIC;
            ES: in std_logic;
            Reset:in std_logic;
           Leer : out  STD_LOGIC;
           c56 : out  STD_LOGIC;
           c64 : out  STD_LOGIC);
end component;

component dna
    Port ( rd : in  STD_LOGIC;
           shift : in  STD_LOGIC;
           clk : in  STD_LOGIC;
            enable: in std_logic;
           dna_data : out  STD_LOGIC);
end component;

component rom
    Port ( clk : in  STD_LOGIC;
            enable : in  STD_LOGIC;
           rom_data : out  STD_LOGIC);
end component;

component scram
    Port ( clk : in  STD_LOGIC;
           datain : in  STD_LOGIC;
            enable: in std_logic;
           dataout : out  STD_LOGIC);
```

```
end component;

signal sA : STD_LOGIC;
signal sB : STD_LOGIC;
signal sEnable : STD_LOGIC; --sC64
signal sClear : STD_LOGIC;
signal sClk : STD_LOGIC;
signal sLeer : STD_LOGIC; -- sRd
signal sShift : STD_LOGIC; -- sC56
signal sDna_data : STD_LOGIC;
signal sEOS: std_logic;

signal sAux: std_logic_vector(20 downto 0);


begin

CLOCK: CLK
  PORT MAP (
    CLOCK => sClk,
       ES => sEOS
  );

Comp: compare
      PORT MAP (
            A => sA,
           B => sB,
           enable => sEnable,
            Clk => sClk,
            Reset => Rst,
            Clear => sClear
            );
Cont: control
      PORT MAP (
            Clk => sClk,
           ES => sEOS,
            Reset => Rst,
           Leer => sLeer,
           C64 => sEnable,
           C56 => sShift
            );

D: dna
      PORT MAP (
            Clk => sClk,
           rd => sLeer,
           enable => sEnable,
           shift => sShift,
            dna_data => sDna_data
            );
R: rom
      PORT MAP (
            Clk => sClk,
            enable => sEnable,
           rom_data => sB
      );

S: scram
      PORT MAP (
```

	Clk => sClk,
	datain => sDna_data,
	enable => sEnable,
	dataout => sA
	);
	
	Clr<=sClear ;
	
	-- Señal de vida, parpadeo de led (T=1,3seg)
	-- Counter
	process (sClk,sAux)
	begin
	if(sClk'event and sClk='1') then
	-- flanco de reloj ascendente
	sAux<=sAux+1; -- cuento uno más
	end if;
	Vida<=sAux(20); -- saco la salida
	end process; -- End Counter
	
	end Behavioral;

Clk.vhd	library IEEE; use IEEE.STD_LOGIC_1164.ALL; use ieee.std_logic_unsigned.all; Library UNISIM; use UNISIM.vcomponents.all; entity CLK is Port (CLOCK : out STD_LOGIC; ES: out std_logic); end CLK; architecture Behavioral of CLK is signal sClk50 : std_logic; signal sClk2 : std_logic; signal sEOS:std_logic; begin ES<=sEOS; STARTUP_SPARTAN6_inst : STARTUP_SPARTAN6 port map (CFGMCLK => sClk50, -- 1-bit Configuration internal oscillator clock output. EOS => sEOS, -- 1-bit Indicates end of startup. CLK => '0', -- 1-bit Input connection to the configuration startup GSR => '0', -- 1-bit Input connection to the global set / reset (GSR). GTS => '0', -- 1-bit Input connection to the global 3-state (GTS) routing. KEYCLEARB => '0' -- 1-bit Clear BBR key when it is set. Note that this signal --

```
--needs to stay low for 200ns (4 clock cycles) to enable
--KEYCLEAR function (to prevent glitches).
      );

-- DCM_CLKGEN: Digital Clock Manager
-- Spartan-6
-- Xilinx HDL Libraries Guide, version 11.2

DCM_CLKGEN_inst : DCM_CLKGEN

      generic map
            (
                  CLKFXDV_DIVIDE => 32,
-- Specifies divide value for CLKFXDV.
                  CLKFX_DIVIDE => 2,
-- This value in conjunction with the input frequency and
-- CLKFX_MULTIPLY value determine the resultant output
--frequency for the CLKFX and CLKFX180 outputs.
                  CLKFX_MD_MAX => 0.0,
-- When using the DCM_CLKGEN with variable M and D values, this
--would specify the maximum ratio of M and D used during static
--timing analysis to ensure proper timing of the DCM output.
                  CLKFX_MULTIPLY => 2,
-- This value in conjunction with the input frequency and
--CLKFX_DIVIDE value determine the resultant output frequency
--for the CLKFX and CLKFX180 outputs.
                  SPREAD_SPECTRUM => "NONE",
-- Specify a supported mode for Spread Spectrum. Must be used -
--in conjunction with the appropriate IP in order to fully
--realize the frequency hopping.
                  STARTUP_WAIT => FALSE
-- Delays configuration DONE signal until DCM LOCKED signal
--goes high.
            )
      port map
            (
                  CLKFXDV => sClk2,
-- 1-bit Divided output clock, Divide value derived from
--CLKFXDV_DIV attribute.
                  CLKIN => sClk50,
-- 1-bit The source clock (CLKIN) input pin provides the source
-- clock to the DCM.In the case of Free-running oscillator
--mode, running clock needs to be connected until DCM is locked
--and DCM is frozen, then clock can be removed. In the other
--modes, a free running clock needs to be provided and remain.
                  FREEZEDCM => '0',
-- 1-bit Prevents tap adjustment drift in the event of a lost
--CLKIN input
                  PROGCLK => '0',
-- 1-bit Clock input for M and/or D reconfiguration.
                  PROGDATA => '0',
-- 1-bit Serial data input to supply information for the
--reprogramming of M and/or D values of the DCM. This input
--must be applied synchronous to the PROGCLK input.
                  PROGEN => '0',
-- 1-bit Active high enable input for the reprogramming of M/D
--values. This input must be applied synchronous to the PROGCLK
--input.
                  RST => '0' -- 1-bit Reset pin
```

	); BUFG_inst : BUFG port map (O => CLOCK, -- 1-bit Clock buffer output I => sClk2 -- 1-bit Clock buffer input); end Behavioral;

Control.vhd	library IEEE;
	use IEEE.STD_LOGIC_1164.ALL;
	use IEEE.std_logic_unsigned.all;
	library UNISIM;
	use UNISIM.VComponents.all;
	
	entity control is
	Port (Clk : in STD_LOGIC;
	ES: in std_logic;
	Reset: in std_logic;
	Leer : out STD_LOGIC;
	c56 : out STD_LOGIC;
	c64 : out STD_LOGIC);
	end control;
	
	architecture Behavioral of control is
	
	signal aux2: std_logic_vector(6 downto 0);
	signal temp, CE, sES: std_logic;
	signal sC56, sC64: std_logic;
	signal sLeer: std_logic;
	type TIPO_ESTADO is (Inicio,
	Lectura,
	Shift,
	Enable,
	Final);
	
	signal estado : TIPO_ESTADO;
	
	begin
	
	-- SRL16E: 16-bit shift register LUT with clock enable operating on posedge of clock
	-- Spartan-6
	-- Xilinx HDL Language Template, version 12.3
	
	SRL16E_inst : SRL16E
	generic map (INIT => X"0001")
	port map (
	Q => sLeer, -- SRL data output
	A0 => '1', -- Select[0] input
	A1 => '1', -- Select[1] input
	A2 => '1', -- Select[2] input
	A3 => '1', -- Select[3] input
	CE => CE, -- Clock enable input
	CLK => CLK, -- Clock input
	D => '0' -- SRL data input
	);

```
   -- End of SRL16E_inst instantiation

--Contador de bits para la señal sShift y sEnable del circuito
completo-----------
process(clk)
begin
      if (clk'event and clk = '1') then
            if temp='0' then
                  aux2<=(others=>'0');
            elsif temp='1' and aux2<"1000000" then
                  aux2<=aux2 + '1';
            else
                  aux2<=(others=>'0');
            end if;
      end if;
end process;


MaquinaEstados :
  PROCESS (Clk, Reset)
  BEGIN

    IF (Reset = '0') THEN
      estado       <= Inicio;

    ELSIF Clk'EVENT AND clk = '1' THEN

      case estado is

        when Inicio  =>
          if ES = '1' then
            estado <= Lectura;
          else
            estado <= Inicio;
          end if;

        when Lectura =>
         if ES = '1' and sLeer='1' then
            estado <= Shift;
          elsif ES = '1' and sLeer='0' then
            estado <= Lectura;
                  else
                        estado <= Inicio;
          end if;

        when Shift =>
          if ES = '1' and aux2="0111000" then
            estado <= Enable;
          elsif ES = '1' and aux2<"0111000" then
            estado <= Shift;
                  else
                        estado <= Inicio;
          end if;

        when Enable =>
          if ES = '1' and aux2<"0111111" then
            estado <= Enable;
          elsif ES = '1' and aux2="0111111" then
```

	estado <= Final;
	else
	estado <= Inicio;
	end if;
	
	when Final =>
	if ES = '1' then
	estado <= Final;
	else
	estado <= Inicio;
	end if;
	
	when others =>
	estado <= Inicio;
	
	end case;
	
	END IF;
	
	END PROCESS MaquinaEstados;
	
	-- Salidas de la máquina de estados
	with estado select
	CE <=
	'1' when Lectura,
	'0' when others;
	
	with estado select
	temp <=
	'1' when Shift,
	'1' when Enable,
	'0' when others;
	
	with estado select
	sc56 <=
	'1' when Shift,
	'0' when others;
	
	with estado select
	sc64 <=
	'1' when Shift,
	'1' when Enable,
	'0' when others;
	
	C56<=sC56;
	C64<=sc64;
	leer<=sLeer;
	
	end Behavioral;

Dna.vhd	library IEEE; use IEEE.STD_LOGIC_1164.ALL; library UNISIM; use UNISIM.VComponents.all; entity dna is Port (rd : in STD_LOGIC; shift : in STD_LOGIC; clk : in STD_LOGIC; enable: in std_logic;

	dna_data : out STD_LOGIC);
	end dna;
	
	architecture Behavioral of dna is
	
	signal sOUT, sDna_data: std_logic;
	signal Q0,Q1,Q2,Q3,Q4,Q5,Q6,Q7,Q8: std_logic;
	
	begin
	-- DNA_PORT: Device DNA Data Access Port
	-- Xilinx HDL Libraries Guide Version 9.1.3i
	DNA_PORT_inst : DNA_PORT
	generic map (
	SIM_DNA_VALUE => X"0167E72508B90FC")
	port map (
	
	DOUT => sOUT, -- 1-bit DNA data output
	DIN => '0', -- 1-bit user data input
	READ => RD, -- 1-bit active high DNA data load
	input
	SHIFT => shift, -- 1-bit active high sift enable
	CLK => CLK -- Clock input
	);
	-- End of DNA_PORT_inst instantiation
	
	process(clk)
	begin
	if Clk'event and Clk = '1' then
	if enable='0' or rd='1' then
	Q0<='0';
	Q1<='1';
	Q2<='1';
	Q3<='1';
	Q4<='1';
	Q5<='0';
	Q6<='0';
	Q7<='0';
	else
	Q0<=Q1;
	Q1<=Q2;
	Q2<=Q3;
	Q3<=Q4;
	Q4<=Q5;
	Q5<=Q6;
	Q6<=Q7;
	Q7<=Q8;
	end if;
	end if;
	end process;
	
	Q8<=sOUT when (enable OR RD)='1' else '0';
	dna_data<=Q1 when enable='1' else '0';
	
	end Behavioral;

Rom.vhd	library IEEE; use IEEE.STD_LOGIC_1164.ALL; library UNISIM; use UNISIM.VComponents.all;

	entity rom is Port (clk : in STD_LOGIC; enable : in STD_LOGIC; rom_data : out STD_LOGIC); end rom; architecture Behavioral of rom is signal sData,aux0,aux1,sD0: std_logic; begin Rom_Data<=sData when enable='1' else '0'; -- SRLC32E: 32-bit variable length shift register LUT -- with clock enable -- Spartan-6 -- Xilinx HDL Language Template, version 12.3 SRLC32E_0 : SRLC32E generic map (INIT => X"49D49836") port map (Q => sD0, -- SRL data output Q31 => aux0, -- SRL cascade output pin A => "11111", -- 5-bit shift depth select input CE => enable, -- Clock enable input CLK => CLK, -- Clock input D => aux1 -- SRL data input); -- End of SRLC32E_inst instantiation -- SRLC32E: 32-bit variable length shift register LUT -- with clock enable -- Spartan-6 -- Xilinx HDL Language Template, version 12.3 SRLC32E_1 : SRLC32E generic map (INIT => X"8D355593") port map (Q => sData, -- SRL data output Q31 => aux1, -- SRL cascade output pin A => "11111", -- 5-bit shift depth select input CE => enable, -- Clock enable input CLK => CLK, -- Clock input D => aux0 -- SRL data input); -- End of SRLC32E_inst instantiation end Behavioral;
Scram.vhd	library IEEE; use IEEE.STD_LOGIC_1164.ALL; library UNISIM; use UNISIM.VComponents.all; entity scram is Port (clk : in STD_LOGIC;

	datain : in STD_LOGIC;
	enable: in std_logic;
	dataout : out STD_LOGIC);
	end scram;
	
	architecture Behavioral of scram is
	
	signal Q1,Q2,Q3,Q4,Q5,sScr: std_logic;
	
	begin
	
	process(clk)
	begin
	if Clk'event and Clk = '1' then
	if enable='0' then
	Q1<='1';
	Q2<='0';
	Q3<='1';
	Q4<='0';
	Q5<='0';
	else
	Q1<=sScr;
	Q2<=Q1;
	Q3<=Q2;
	Q4<=Q3;
	Q5<=Q4;
	end if;
	
	end if;
	end process;
	
	sScr<=Q4 xor Q5;
	
	Dataout<= sScr xor Datain when enable='1' else '0';
	
	end Behavioral;

Compare.vhd	library IEEE; use IEEE.STD_LOGIC_1164.ALL; use IEEE.NUMERIC_STD.ALL; library UNISIM; use UNISIM.VComponents.all; entity compare is Port (A : in STD_LOGIC; B : in STD_LOGIC; clk: in std_logic; enable : in STD_LOGIC; Reset: in std_logic; Clear : out STD_LOGIC); end compare; architecture Behavioral of compare is signal sComp, aux0,aux1: std_logic; begin process(clk,reset)

	begin
	if Reset='0' then
	sComp<='0';
	aux0<='0';
	aux1<='0';
	elsif Clk'event and Clk = '1' then
	aux0<=enable;
	
	if enable='1' and aux0='0' then
	sComp<='1';
	aux1<='1';
	end if;
	
	if aux1='1' and enable='1' then
	if A/=B then
	sComp<='0';
	end if;
	
	end if;
	end if;
	end process;
	
	Clear<=sComp when (aux1='1' and enable='0') else '0';
	
	end Behavioral;

Tb.vhd	LIBRARY IEEE;
	USE IEEE.STD_LOGIC_1164.ALL;
	USE IEEE.STD_LOGIC_ARITH.ALL;
	USE IEEE.STD_LOGIC_UNSIGNED.ALL;
	use work.deftipos.all;
	entity tb is
	end tb;
	
	architecture a of tb is
	-- Asignacion constantes:
	constant semiperiodo: time:=10ns;
	
	-- Declaracion senales:
	signal clk: std_logic := '1';
	signal es,st,aleatoria: std_logic;
	signal s_n: std_logic_vector (5 downto 0);
	signal fin: std_logic;
	signal value_test_9,a: integer;
	signal mem: std_logic_vector(399999 downto 0);
	signal m:std_logic_vector(15 downto 0);
	
	--Declaracion componente
	component test_9_2
	port (clk : in std_logic;
	Reset : in std_logic;
	start : in std_logic;
	aleatoria_test9: out std_logic;
	--si secuencia es aleatoria
	value_test_9: out integer;
	fin : out std_logic;
	--cuando la haya calculado
	s_n : in std_logic_vector (5 downto 0));
	end component;

```
begin

uut: test_9_2
                    port map (clk=>clk,eos=>es,start=>st,
aleatoria_test9=>aleatoria,value_test_9=>value_test_9,fin=>fin,
s_n=>s_n);

clk <= not clk after semiperiodo;
mem<="11000110…… 01010101010101010101";--400000 bits
process(clk,ES)
begin
    if ES = '0' then
            s_n<=(others=> '0');
            m<=(others=> '0');
            m(0)<='1';

      elsif clk'event and clk = '1' then
            if st='1' then
                  if m <= Bl then
            m<=m+'1';
            s_n<=mem((6*a-1) downto (6*a-6));

                  end if;
            end if;
      end if;
end process;

a<=conv_integer(m);

PROCESS
BEGIN

   ES<='0';
   st<='0';
   WAIT FOR 2*semiperiodo;
   ES<='1';
   WAIT FOR 8000*semiperiodo;
      st<='1';
   WAIT FOR 4000000*semiperiodo;

END PROCESS;
end a;
```

deftipos.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
package deftipos is
  type matriz_log_pos is array(1 to 136) of integer range 0 to
63304;--indices donde cambia el valor del logaritmo
  type matriz_log_cont is array(1 to 136) of integer range 0 to
160;--valores de logaritmo para cada inddice especificado en
pos

  constant Smin   : integer:=3327377;--f_n<8.7282-secuencia
aleatoria
  constant Smax   : integer:=3351292;--f_n>5.6391-secuencia
aleatoria
  constant K      : integer:=64000; -- bloques para analizar
```

```
  constant Q      : integer:=640;--bloques iniciales
  constant Bl     : integer:=64640;--bloques totales K+Q
  constant LOG_POS        : matriz_log_pos:=
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,20,21,22,24,26,27
,29,31,34,36,39,41,44,47,51,54,58,62,67,72,77,82,88,94,101,108,
116,124,133,143,153,164,175,188,201,216,231,248,266,285,305,327
,350,375,402,431,462,495,531,569,609,653,700,750,804,862,923,99
0,1061,1137,1218,1306,1399,1500,1607,1723,1846,1979,2121,2273,2
436,2611,2798,2999,3214,3445,3692,3957,4241,4545,4871,5221,5596
,5997,6428,6889,7384,7913,8481,9090,9742,10442,11191,11994,1285
5,13778,14767,15826,16962,18180,19484,20883,22382,23988,25710,2
7555,29533,31652,33924,36359,38968,41765,44763,47976,51419,5510
9,59065,63304
);--posición apartir de la cual hay una serie de valores
identicos de logaritmos

constant LOG_CONT       :
matriz_log_cont:=(0,10,16,20,23,26,28,30,32,33,35,36,37,38,39,4
0,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,6
1,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,8
2,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,100,101,10
2,103,104,105,106,107,108,109,110,111,112,113,114,115,116,117,1
18,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,
134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149
,150,151,152,153,154,155,156,157,158,159,160
);--valores de logaritmos para cada posicion
end deftipos;
```